UNIVERSITY
of
GLASGOW

Department of
Computing Science

Ph.D. Thesis

Use of Preemptive Program Services with Optimised Native Code

Antonio Cunei

Submitted for the degree of
Doctor of Philosophy
at the University of Glasgow
August 2004

Typeset on 18th August 2004.

This Ph.D. thesis was prepared using the LaTeX typesetting system, and composed using the LaTeX-oriented open source editor LyX. The style file used for typesetting was originally customised by Tony Printezis. The fonts used are Times for standard text, AvantGarde for the chapter headings, Helvetica for section headings and page numbers, `Courier` for the code, and *ZapfChancery* for the quotations.

Nothing worth doing is easy.
— Old adage

Abstract

In modern computer systems, the normal execution of a thread may be temporarily suspended to allow a service routine to manipulate the thread memory and state. For instance, a garbage collector could modify the content of the heap to free up unused memory, or a migration mechanism could extract the thread state and move it on a different machine. The execution of those service routines may either be visible from the user program or it might happen in a transparent way, depending on the techniques used and on the implementation.

If a service routine can intervene in the normal program execution in a preemptive fashion, the normal thread activity may be interrupted, potentially, at any point. If optimised native code is being executed, the thread suspension could happen at any arbitrary machine instruction. However, the intervening routine could need access, to perform correctly its functions, to some specific information that in general might not be available at that particular stage during execution. For instance, a memory block can only be moved in the heap if the location of all the pointers that refer to that block is known, so that those references can be updated when the block is relocated. Increasing the availability of that kind of information, therefore, can be of great help while designing a system that should offer those services preemptively.

In general, little literature exists about techniques able to support different kinds of service routines in optimised native code and to obtain the necessary information for every machine instruction. Some work exists on garbage collection at every machine instruction, some work discusses the extraction of the necessary information when using compiled code, but no systematic discussion is apparently available about supporting multiple service routines using preemption and native optimised code. This thesis discusses the prerequisites of such a support, the technical challenges and the techniques, some known, other newly developed in the context of this research, which can be used to develop such a system. The development of a test implementation, showing the usefulness of those techniques and ideas, is also described and the experimental results are discussed.

'tis the advisor who suffers from bad advice.
— Anonymous

Acknowledgments

Many people have helped me throughout the development of this Ph.D. research. First and foremost I would like to thank my current supervisors, Prof. David Watt and Dr. Simon Gay, for their advice, their insightful suggestions, and their infinite patience. I would also like to thank my former supervisors, Prof. Malcolm Atkinson and Dr. Tony Printezis, for their encouragement, support, and comments during the first phase of this research. Many other people contributed ideas and personal knowledge towards the refinement of the techniques described in this thesis.

My internships in Sun Microsystems Laboratories proved invaluable, allowing me to obtain a more complete view about the problems involved in garbage collection algorithms, and some obscure details of SPARC microprocessors and system architectures in general. For making it such a great experience I am grateful to Mario Wolczko and Greg Wright, who offered plenty of help and precious comments. Ross Knippel, David Cox and Chuck Rasbold kindly took the time to explain to me many details about the inner workings of the JBE Java compiler, which eventually became part of Sun's ExactVM.

Amer Diwan clarified some crucial details in the implementation of garbage collection maps with an experimental customisation of Modula-3 on GCC, originally for the VAX, and patiently dealt with my nagging requests for more information. Tony Hosking offered to retrieve a copy of the source code of a subsequent adaptation of the same project to the RISC architecture. Alex Garthwhite offered important information and bibliographical references about thread-local heaps. Dianne Ellen Britton kindly retrieved a paper copy of her '75 Master thesis on automatic heap management for the language Pascal and sent me a digitised version of her work. It is surprising to discover how certain issues related to pointers handling and initialisation have changed very little since then. Richard Hudson helpfully pointed out the difficulties of tracking pointers while using write barriers. John Reppy spent quite some time explaining to me a great deal of important details of the implementation of MOBY, pointing out useful additional references. Eliot Moss shed some light on the inner workings of the Trellis/Owl system. David Ungar suggested ways to improve the testing of the prototype, and gave me a whole new light in which to consider this research. A whole new light on research in general, actually. Jim Stichnoth clarified the handling of system code in their Java compiler system. Reinhard Wilhelm offered precious references on static analysis techniques.

The development was conducted using a number of programming environments, including the Linux operating system, Sun Microsystems Solaris, the Cygwin environment and the Simics simulation suite, a commercial tool generously made available for free by Virtutech to students and academic institutions. The main body of research was developed using the GNU Compiler Collection, and debugged using GDB and Insight. Insight. All trademarks, product names and company names or logos used in this thesis are the property of their respective owners. To the best of the author's knowledge, this thesis contains no material previously published or written by other persons, except where due reference is made.

This work was made possible by a University of Glasgow Postgraduate Scholarship and by research funds offered by Prof. Malcolm Atkinson, in the context of the grant arranged by Dr. Mick Jordan of Sun Research Labs (Palo Alto) as part of the Sun collaborative research agreement to investigate persistence for Java.

Faber est suae quisque fortunae.

— ***Appius Claudius Caecus**, Roman orator*

(Each man is the smith of his own fortune.)

The most merciful thing in the world... is the inability of the human mind to correlate all its contents.
— ***H. P. Lovecraft**, 1890 - 1937*

Contents

What I give form to in daylight is only
one per cent of what I have seen in darkness.
*— **M. C. Escher**, 1898 - 1972*

List of Figures

Science may set limits to knowledge, but
should not set limits to imagination.
*— **Bertrand Russell**, 1872 - 1970*

List of Tables

Chapter 1 gives a general introduction to the context of this work. After defining the term "program services", the chapter describes the way in which they can be used preemptively, and gives a general overview of the issues involved.

Chapter 2 explains in greater detail the problems involved in supporting the preemptive use of program services and explains how the main issue is tracking the types of data during execution. In many cases, finding where the pointers are is sufficient. The various technical challenges are listed and discussed.

Chapter 3 introduces some techniques that can be used to address the challenges described in the previous chapter. A more precise description of the working context is given, and the concept of "mode" is introduced. Particular features of compilers and microprocessors that can introduce additional complexity are discussed. The content of the following chapters is introduced and discussed in general terms. The experimental prototype, described in Chapter 9, is also introduced.

Chapter 4 focuses in more detail on determining where the pointers are in the registers of the microprocessors, while the code is running. The usefulness of a customised liveness analysis, described in detail in the following chapter, is explained.

Chapter 5 is devoted to a formal discussion of the customised liveness algorithm, and the related verification mechanisms that can be used to ensure the consistency of the information obtained from the compiler. The implications of the use of delay slots are discussed, and the way in which the liveness algorithm and the checks can be adapted is explained.

Chapter 6 analyses the mechanisms that can be used to discover where pointers are on the stack. The various components of the stack frames are explored individually, and the particular technical problems involved are discussed for each of them.

Chapter 7 discusses the discovery of pointers in the heap. The facilities that should be made available by the heap handler are discussed.

Chapter 8 contains a description of the operations that should be performed at runtime, when a program service is requested preemptively. The dynamic aspect, described in this chapter, complements the mostly static analysis described in the previous chapters.

Chapter 9 presents an experimental prototype that was used to highlight possible implementation difficulties. The structure of the prototype, its implementation, and the results obtained are analysed.

Chapter 10 discusses the particular problems related to the use of "derived pointers". Existing techniques and a new proposal to deal with them are presented.

Chapter 11 presents a real-life example of a problem in which the use of preemptive program services could be beneficial. In particular, the techniques described in this work could be used to implement thread-local heaps while using preemptive thread switching.

Chapter 12 contains a final summary, an evaluation of the technique, and a conclusion.

Look with favour upon a bold beginning.
*— **Virgil**, 70 BC - 19 BC*

Chapter 1

Introduction

1.1 Program services

On modern, complex computer systems it may be useful, or even necessary, to manipulate the state of live, running threads and their data. For instance, if the program is long-lived, the ability to perform dynamic reallocations of threads on different CPUs, or to different nodes in a network, can allow for a considerably more efficient use of processing power and may be crucial in preventing the system from exhausting the available resources. Similarly, the ability to save the thread state, including the live data, at periodic intervals can be very useful in case of hardware failures, power cuts, etc. in that the state can be subsequently resumed and restarted, therefore preserving intermediate results previously computed up to the last commit point.

We will refer with the term "service" to any facility that temporarily suspends the normal program execution to intervene on its memory, and more generally on its state, in order to improve the normal program operation or to offer additional features. Garbage collection, persistence and migration are some of the services that rely, for their correct operation, on the ability to access the internal thread structures and data at certain moments during execution. However, not all the implementations allow a service to operate at any given time. A garbage collector could require, for instance, the collection to be performed only at specific moments during the execution of each thread, requiring all the threads to reach a safe point before proceeding.

An analysis of some typical implementations will reveal a range of possible levels at which the service can be implemented. If a fully interpreted virtual machine is used, for instance, the virtual code will usually contain no indication of the occurrence of garbage collection, which will be performed automatically by the virtual machine, instead. If a Just-In-Time compiler is used, the services will typically be performed, instead, by calls inserted in the compiled code, in selected garbage collection points, while the original source code does not contain any indication of the operation taking place. Still another approach could be maintaining the operation explicit up to the level of the user code, requiring the programmer to place explicit calls to the service routines in the user program.

Each of the different levels offers a “transparency” of the service above a certain level of abstraction, while the underlying layers must deal explicitly with the service invocation. Each level of abstraction is also related to a certain degree of granularity, which determines how often the operation can take place, and the minimal interval, in terms of time or number of primitive instructions, that separates two possible occurrences of the service invocation. The minimum delay between two operations also influences the latency, that is the time necessary for the system to respond (invoking the service) when an event requesting the operation is received. For example, if a power failure is detected and the state of the running thread must be saved to disk as soon as possible, a low latency might be crucial for correct operation. If the guaranteed maximum latency is too long, or does not have a known upper bound, the save operation can be delayed excessively, and some of the system state can be lost. Having a guarantee about the maximum latency for a specific service can be crucial in real-time applications.

1.2 Machine instructions and preemption

The smallest granularity in the frequency of service invocation that can be achieved on common computer systems is that of the single machine instruction. If the required support for a certain service can be offered preemptively, the program could be temporarily suspended at any machine instruction and its internal state manipulated appropriately. An advantage of such an approach is the absence of additional code that, using other techniques, is usually inserted to mark a possible point of execution of the system routine (safe point, yield point and so on). The normal execution can therefore proceed at full speed, and greater are the opportunity to perform code optimisations, for instance using peephole optimisers [ASU86, WG95], since the flow of instructions is not interrupted by additional instructions or function calls.[1] Another advantage, compared to additional instructions inserted in the normal code, is the simplification of the program logic, since the service will operate transparently. The programmer does not need to write additional code, or to be concerned with the details of the services in use. A third advantage, as mentioned, is the much finer granularity and consequently the potential for lower latencies.

The technique appears to be quite appealing, but implementing the desired support is not straightforward. The system must ensure proper operation for the necessary state manipulations at any arbitrary point of the program, even if the program is in that moment engaged in complex memory operations, system routines or other activities that could interfere with the correct handling of the service. For instance, if the program is interrupted in the middle of a memory allocation, the internal structures of the heap could be left in an inconsistent state while a garbage collection is attempted [SCM99]. Similarly, input/output done using Direct Memory Access, using a block on the heap as the buffer, would cause data corruption if the block in question is moved half way through the data transfer.

[1] Safe points can also be implemented using different solutions, although such approaches are less common. For instance, it is possible, when a request for a service is received, to replace on-the-fly some instructions in the normal code with a special handler [Age98]. Conceivably a hardware breakpoint could also be used in a similar way. Certain issues involved in the mechanism have some similarities with those related to preemptive services, as will become apparent progressing in the discussion.

On the other hand, taking into account the noted exceptions, being able to perform an exact garbage collection, a system checkpoint, or a thread migration at any assembler instruction would mean being able to use fully optimised code, reducing to the minimum the latency required for the activation of the service routine, and in general having a much finer control over the use of system resources. For example, in multithreaded environments, if preemptive thread scheduling is used, performing a "stop the world" exact garbage collection (GC) implies rolling forward each of the running threads to their next point in the code in which GC can be safely performed, which might involve a substantial delay. Being able to perform a garbage collection at any instruction, for each thread, would reduce, in this case drastically, the necessary delay.

The literature on preemptive garbage collection, or other heap manipulations, is often vague, extremely sparse, and the relevant projects are usually limited to a single programming language and operating environment. This work, conversely, focuses on a more complete view of the general problem, and on a detailed analysis of the challenges and techniques surrounding the subject. An in-depth analysis of the many hidden implementation difficulties and of the technical solutions involved, some of which are believed to be presented here for the first time, is also included. A list of available literature, related to the topic, will be presented in Section 3.1, but it is useful to introduce immediately a useful mechanism, often used in previous works.

A possible technique to implement garbage collection, or other services, in a preemptive way is the use of maps, built statically, which associate values of the program counter (PC) to some information for that point in the compiled code. Such maps are variously mentioned in literature with the names "PC maps", "trace tables", "GC maps", and similar terms. PC maps can be used to describe information about the pointers contained in registers and/or in the stack at particular points in the code. If the PC maps are defined over every instruction, they can be used to the effect of making every instruction a potential safe point. In other words, the corresponding code could be regarded as "safe code", in which an interruption and the corresponding service can take place at any time. In a sense, safe code is the counterbalance to critical sections, in which a service routine must not interrupt the normal flow of code execution.

In general, there seems to be a lack in literature of a systematic study of techniques able to offer a generic support for different services, in as wide a context as possible. The main apparent obstacle to such an effort is the supposed excessive size of the tables needed to maintain enough information, in correspondence to every assembler instruction, to implement the required support. Such an argument, however, appears to be considerably less relevant today than it used to be in the past, thanks to the considerable availability of memory on modern machines. While some literature is available with regard to garbage collection, it is extremely difficult to find any examples of previous work on preemptive migration or checkpointing at any machine instruction.

Exploring the feasibility of checkpointing, garbage collection and migration at every assembler instruction makes, consequently, for a remarkably interesting research topic, which could prove useful to improve performance and flexibility of modern computing systems. In this work new techniques suitable to implement such a generic support, and improvements on existing ones, will be explored, and an implementation of some of those techniques will be shown. The discussion will revolve around compiled languages, either statically or just-in-time, but the same mechanisms could also be applied to interpreters, interrupting preemptively the code of the interpreter.

It should be noted that the kind of support needed would be similar for other forms of memory manipulation or program state inspection, and the work can be therefore extended in other areas as well. In particular, the additional information about the compiled code could also be used by debuggers in order to present a more accurate representation of the running program. For example, in conventional debuggers the relationship between the original source code and the resulting assembler is often not clear, especially if the resulting code is heavily optimised. Stepping through optimised code, therefore, it is often not possible to determine the effect of the individual instructions of the source program. At the same time, stepping through individual assembler instructions is of little use, since there is not enough information useful for debugging available at that level. In particular, it is not known, in conventional debuggers, how the individual machine registers relate to the values manipulated in the source program at each assembler instruction. Being able to associate more information with each assembler instruction, consequently, would allow the programmer to step through the compiled code having more information about the manipulated values, and having therefore a better picture of what is really done by the executed code.

The rest of this work will mainly focus on garbage collection, persistence, and migration. The next section will review more closely the issues involved in offering each of the mentioned services at each machine instruction.

1.3 The issues involved

1.3.1 Garbage collection

Garbage collection involves the automatic removal from the heap of some blocks of memory that will no longer be used, thus freeing heap space for further allocations. In general, it is not possible to determine whether an arbitrary block of memory will really be used again, since that would imply some knowledge about the future behaviour of the running program [ADM98]. It is however possible, in general, to determine that certain heap blocks can no longer be reached by the running program. The unreachable blocks can then be safely disposed, and their memory freed. If the garbage collector is able to identify the location of all the pointers used by the program, the full set of unreachable memory blocks can be disposed. In that case the garbage collector is usually said to be exact, or accurate, although its ability to reclaim unused memory is lower than that of an ideal, fully precise collector. For an interesting comparison of exact versus ideal garbage collectors, see Shaham et al. [SKS00].

If, conversely, it is not possible to discriminate whether some of the values used by the program are pointers or simple scalar values, it is necessary to consider that those values “might” be pointers, adopting a conservative approach, and possibly not freeing all of the unreachable memory [BW88]. An exact garbage collector allows for a better management of system resources, since a greater amount of available memory can be reclaimed during every collection. Also, knowing the location of all the pointers allows the memory blocks to be relocated in the heap, using compaction strategies to reduce heap fragmentation. For certain low-power devices, memory compaction can also be used as a strategy to save battery power, since the memory banks not currently in use can

be switched off.

Another relevant aspect of a garbage collector is its support for preemptive operation. While a non-preemptive approach usually relies on garbage collection safe points, or on maintenance routines invoked periodically by a virtual machine, implementing an exact garbage collector that can execute at any point in the compiled code is usually more complex. Using selected garbage collection safe points all the registers can be flushed to memory before calling the garbage collector, and it is consequently enough to keep track of the pointers present in the heap and on the stack. Conversely, if the program can be interrupted at any machine instruction, some of the pointers might be temporarily cached in registers. Those registers are, by all effects, valid pointers to heap objects, and must be taken into account while performing the collection. A similar problem exists if safe points are used, but registers are not flushed to memory.

A conservative approach could consider all the registers as possibly containing pointers, but an improved collection could be performed if it were possible to know exactly, for every value of the program counter, which registers contain pointers and which just scalar values. Furthermore, if the garbage collection requires the memory blocks in the heap to be moved, knowing which registers contain pointers is essential, since their content may need to be properly adjusted. A simplified approach could be to subdivide the set of user-accessible registers into two sets, one of which is reserved for pointers. However, that solution would severely limit the ability of the compiler to reuse registers while generating the code, and might lead to less efficient code. If fully optimised code is desired, the only viable option is somehow determining the life of pointers in registers throughout the code.

1.3.2 Data persistence

Making data persistent, so that it outlives the program that created it, means essentially saving a copy of the data itself from memory onto some sort of storage support, in a format that allows a subsequent retrieval and reuse. In the transparent forms of persistence, however, the programmer does not have to worry about explicitly making the data persistent. The task is instead delegated to the underlying system, which will perform the operation periodically, in a completely automated manner. Such an approach is especially interesting when coupled with orthogonal persistence, in which all data is treated in the same way regardless of type, and every memory object has equal right to persistence. Implementing transparent orthogonal persistence means that the programmer not only does not need to be concerned with the details of making the data persistent, but does not even need to invoke explicitly any particular routine for the operation to be performed. All of the persistence-related operations will be therefore entirely transparent to the programmer, who can assume that every data structure created will be managed and saved in a completely automated way.

If the level of granularity considered is that of the single machine instruction, the system should be able to handle the transfer of data from memory to disk and vice versa without requiring any additional embedded code, with respect to what would normally be generated by the compiler, while still being able to manipulate memory as required. All of the required data transfer operations (eviction and transfer from memory to storage, and successive reload) must be therefore

transparent. A transparent loader on demand, controlled by the system using native hardware facilities, can take care of moving objects from the persistent store to memory, converting as necessary the internal object representation. Evicting objects from memory and converting them to their persistent form can be done at any machine instruction if the set of all the pointers is completely known, so that all the references involved (both in the object being evicted and in objects that refer to it) can be transformed into the corresponding persistent form.

While some implementations exist of the kind of transparent loader described (used, for instance, by the Texas persistent store to implement pointer swizzling at page fault time [WK92]), no examples appear to be available in literature of a checkpointing in which the set of all the pointers in use is known at every instruction. The only solutions that implement a preemptive checkpointing imply instead either a snapshot of the entire memory [SSF99], or non-relocatable memory blocks, as used by Single Address Space Operating Systems (Sombrero, Mungi, Opal, for instance [SM98a, Voc98, CLBHL93]). It should be noted that Single Address Space systems, tying permanently every object to a virtual address, may prevent an efficient re-clustering of objects into pages, which could potentially also lead to a less efficient use of the processor cache.

Once again, as in the case of garbage collection, the crucial point appears to be the availability of full pointer information for every machine instruction, including the contents of registers, so that the memory objects can be treated not just as byte sequences, but as more abstract and structured entities.

1.3.3 Data migration

The case of data migrations is very closely related to that of data persistence, in that both techniques require enough knowledge about the memory objects to be able to move them freely in and out of the physical memory while maintaining the objects' identities unaltered, and the program logic unchanged. Knowing the location of the pointers in memory is sufficient to move or copy the objects to another machine, at possibly different addresses, while still preserving the logical interconnection between them dictated by the original pointers. Typically, if the hardware architectures of the two machines exchanging the data are the same, no other conversion is necessary, and the requisites are therefore not different from the case of data persistence.

However, the matter is slightly more complicated if the two machines involved in the migration are based on microprocessors belonging to different families. In that case the original, native format of the transferred object could not correspond to the native format on the destination machine. The most obvious problem is endianness. If the source machine is big-endian and the target one is little-endian, for example, the format of all multi-byte values will need to be rearranged to conform to the new native format (in the hypothesis that both machines use native, optimised code to manipulate data objects). Another aspect that needs to be taken into account is memory alignment, and the possible location of padding values. Not all machines can or are equally efficient in accessing long values aligned to different memory boundaries, and the Application Binary Interfaces of each suggests, usually, standard ways to align data, sometimes in mutually incompatible ways across microprocessor families. Therefore, the internal structure of the data object will need

to be opportunely rearranged, and the complete internal structure of the transferred object needs to be known, including the location of all the pointers and the precise length and type of every component. The exact details of the conversion, when and where it is performed, will then depend on the specific implementation choices, and a variety of possible solutions can be adopted (for instance converting to a neutral format, or sending as-is and converting on the destination machine, or converting in the target format before sending, and so on).

The crucial point, however, is that the data object must be considered, during the migration, as having a defined and known structure so that it can be treated at a more abstract level, in a manner that is independent of the specific architectures involved. It is describing the abstract format of the object in terms of the common features among the different architectures that enables us to perform the necessary conversions. Very similar considerations will be repeated, in a more articulated way, for the case of heterogeneous thread migration, in Section 1.3.5.

1.3.4 Thread persistence and migration, same architecture

Making a thread persist, or migrate, implies extracting its state and converting it into a format such that a subsequent resumption, on the same or a different machine, is possible. A proper resumption of the thread is usually only possible if the thread still has access to the local environment, including all the accessible heap objects, which were present at the moment of the state extraction. Accordingly, making a thread persistent or migrating it on another machine implies obtaining from the system enough information (program counter, stack, heap objects and so on) to restore the running program at a later stage.

As described for the previous techniques, there are several levels of abstraction, and related degrees of granularity, that can be used. For instance, if a virtual machine is used, the virtual program counter, the virtual machine stack and some additional state will usually be the components, together with the reachable parts of the heap, of the needed state. If the desired degree of granularity, on the other hand, is that of the single machine instruction (which brings all the advantages previously discussed), the necessary thread state will be typically composed by the microprocessor internal state, the stack used by the thread and the state of the heap.

As previously described for data migration and persistence, considering those elements from a more abstract point of view allows for a more flexible manipulation. Moving the state from one machine to another (as in checkpoint/restore, or migration) is however possible only if enough information for a meaningful conversion is available. In the case of a Single Address Space system, the source and the destination have the same structure, and no conversion is required. The obvious drawback is that no object can be relocated in memory, which means that compaction is not an option. If source and destination have the same hardware structure, but memory blocks need to be moved, the more convenient abstraction could be considering the pointers as abstract entities, while the remaining data is left unchanged. As previously detailed, knowing the set of all the pointers in use (in the heap, in the microprocessor registers, on the stack) for every value of the program counter is enough to perform all the required manipulations. Consequently, the kind of support previously considered for data persistence, garbage collection, and data migration is, in principle, enough to enable thread migration and persistence if a single architecture is used. On

the destination machine, the state will be adapted modifying the pointers as necessary, be they in the heap, on the stack or in system registers, after which the thread can resume.

Some difficulties may arise, however, if the mentioned support is not system wide, and part of the system does not conform to the abstraction required. In that case, part of the meaningful state of the running thread could be external to the program domain, and preserved in system structures (open files, network sockets etc.) The problem can be alleviated by keeping a cached copy of the system state in local structures, which can be saved with the rest of the state. In an ideal case, the best solution would be to have the mentioned support for persistence and migration extended to the entire system, so that the full thread state can be manipulated at any moment.

1.3.5 Thread persistence and migration, heterogeneous architectures

If the source and the destination machines use microprocessors that belong to different families, a much more sophisticated approach must be adopted. The common elements between the two (or more) architectures, in this case, will be fewer. A conversion of data structures, as mentioned in the section on data migration, will be necessary. Additionally, the stack and the microprocessor registers contents will have to be transformed from one machine to another, possibly completely different, allowing the thread to resume seamlessly, which is definitely not a trivial task. To overcome the difficulties, most of the systems which allow thread migration in heterogeneous environments adopt a higher level of abstraction, for instance moving the state of a virtual machine, or restricting the opportunities for checkpointing/migration to selected points. The latter approach reduces the granularity with which the operation can be performed, and does not offer all the advantages of full preemption, explained in Section 1.2.

The ideal approach would be to have the ability of freezing a thread preemptively at an arbitrary point in the native, fully optimised compiled code, while still being able to obtain enough information about the state in that point of the code to allow for a conversion of the state in a format suitable for another machine with a different microprocessor. There seems to be, at the moment, no literature at all on techniques that could allow such a transformation to occur arbitrarily, for any given value of the program counter.

The key, as can be inferred by the previous discussion, can be the use of the common elements between the programs compiled for the two different architectures, so that a common format can be found. In particular, producing the code for two machines from the same source code by means of a compiler involves the use of a common representation of the program within the compiler itself. Such an intermediate representation could be an interesting candidate for the kind of transformations required.

1.4 Summary

- What is needed to support services at any machine instruction:

  An improved support for garbage collection, persistence and migration at every machine instruction on a single architecture can be obtained if it is possible to generate enough

information to associate every value of the program counter with the set of all the pointers used at that point. The pointers can be contained in the heap itself, in the stack or in the microprocessor registers (the global data area can always be considered as a special kind of activation frame, or possibly as a special heap block). That kind of support can be combined with a transparent loader, guided by the memory management unit, to make the memory manipulations entirely transparent to the user program.

Support for heterogeneous migration can be added if a proper abstract representation can be found for data objects and the thread state, together with a convenient way to convert the native formats of the various architectures in one another, even when the state is extracted at an arbitrary point in the code.

- What has been done in literature:

  Generating tables containing the required information (the location of all the pointers) for every value of the program counter could have been regarded in the past as impracticable due to the memory requirements. Thanks to the capabilities of modern computer systems, the technique appears nowadays considerably more appealing. Some work has been done to support garbage collection at every machine instruction in the context of specific systems, but no literature is available on generic support for multiple services, usable in a preemptive fashion. Hardware-supported loaders are conversely commonly deployed in virtual memory subsystems, persistent systems, etc., and their use is essentially orthogonal to pointer tracking, therefore the two problems can be treated independently.

- The purpose of this work:

  – Studying in more detail, and in a more comprehensive light, the techniques that can be used to generate automatically the data structures necessary to offer the described support.
  – Defining a more integrated approach, so to offer GC, persistence and migration at every machine instruction in a simple and consistent way.
  – Collecting the sparse knowledge on the subject, adding original contributions. Evaluating the complexity of the approach and its usefulness.
  – Offering an insight into the concrete implementation challenges of a complete system, describing the working steps, the required techniques, and the hidden problems, also thanks to the development of a concrete prototype, which will be used to expose subtle, but important, details not usually documented.

The remainder of this thesis will be devoted to achieving these goals.

```
#define QUESTION ((2b) || (!2b))
/* Shakespeare */
```

Chapter 2

Focusing on the Problems

2.1 Requirements

As the previous discussion shows, offering the desired support for garbage collection, persistence and migration (data and threads) at every machine instruction, in a completely transparent way, involves the following:

- being able to track the pointers (in the heap, in the stack, and in the microprocessor registers), so that every heap block can be relocated or evicted from memory at every point in the code while the references to that block are correctly updated,
- implementing a transparent loader, so that an attempt to access a portion of memory previously evicted can result in an automatic reload,
- devising a way to convert the extracted microprocessor state, the stack, and the relevant portions of the heap in different formats, suitable for use with a persistent store or for migration. The conversion can be particularly complex if the machines involved use different microprocessors.

It is now time to determine more precisely which techniques can be used to fulfill those requirements, and which unsolved problems remain, requiring therefore the introduction of new techniques.

2.2 From the problems to the techniques

A transparent loader can be implemented in a rather straightforward way using the Memory Management Unit to trap memory accesses to certain areas of the address space. When accesses to those areas are detected, an exception can be raised, giving control to the loader. The loader

can then load the necessary pieces of data and adjust the memory configuration as needed before returning the control to the thread which was previously running. Examples of such loaders are available in every virtual memory subsystem, and the techniques are well understood, so they will not be further discussed here. It should be noted, however, that the ability to find all the pointers can significantly enhance the capabilities of a conventional loader, allowing memory blocks smaller than a single page to be easily loaded and evicted individually, if needed, or re-clustered into memory pages depending on dynamic usage patterns.

The techniques used to convert memory blocks into a persistent form and vice versa are similarly well known, although not generally used in conjunction with the level of granularity considered here, and numerous systems offer solutions for orthogonal persistence, in various forms. The format conversion, as long as a single hardware platform is involved, consists essentially of the replacement of pointers to memory blocks with more abstract Persistent Identifiers. The association between the two representations is then preserved into a central repository, usually referred to as Resident Object Table (ROT). More details can be obtained from the very rich literature that exists on the topic, related to persistent object stores, persistent systems, and so on [ABC+83, AM95, HC99b, DRH+92, SSF99, Kak98].

On the other hand, as previously mentioned, very little research exists on finding pointers (and more generally determining data types) in compiled and optimised code at every machine instruction. No working solution seems to be available for the migration of threads between heterogeneous architectures using native code, using on-the-fly transformations on the relevant data structures. The latter problem is essentially equivalent to make data and thread state persistent on one system and restoring the saved information on another machine with different hardware architecture.

While data conversion can be accomplished by adapting the internal representation of data contained in the heap into the desired native format of the destination machine, which is not terribly difficult to do as long as the structure (type and size of every component) of each migrated memory block is known, converting in a suitable way the stack content and, more crucially, the internal state of the microprocessor, is considerably more difficult. While an object allocated in the heap does not usually change its structure during its lifetime, types and contents of data contained in stack and registers can change much more often, possibly at every machine instruction. The state of the microprocessor, additionally, may not have a direct equivalent on the target machine.

Summarising, the most crucial issues to solve, in order to complete the required support at every machine instruction, are type tracking and state conversion, each of the two for heap objects, stack contents and machine registers. To organise the following discussion, it can be useful to consider again the differences that exist between performing operations in the context of a single architecture and using instead a heterogeneous environment. Each of the two cases requires a different level of abstraction and a different abstract representation, suitable for the kind of conversions that are needed.

If a single architecture is involved, the data and thread manipulations involved preserve the format of all the scalar values used, requiring only the pointers to be converted or adjusted, following object eviction, loading or relocation. There is no need to change the stack layout, and only the pointers, among the values held in stack or registers, may need to be changed. In other terms, the

abstract format required only needs to represent in an abstract form what may change between the different contexts. Tracking the type of the data in use will therefore consist in tracking the pointers. Since no conversion is performed on scalar values, no detailed information about their representation is ever required. Similarly, the necessary conversions will only involve changing the pointers into persistent identifiers and vice versa, or adjusting pointers, but no other transformation on the remaining data will ever occur. Also, the code executed will remain unchanged, unless absolute cross-references to code locations (absolute jumps, for instance) are used.

Conversely, if multiple architectures are involved, the abstract format used will have to rely on the common characteristics of the different machines. Generally speaking, nearly all of the microprocessors currently on the market have broadly similar structure and behaviour, and one may safely assume that all of them have a natural representation for 8-bit, 16-bit, 32-bit, 64-bit (and wider) integers, IEEE754 floating points and pointers. An abstract representation for the data will therefore describe in abstract terms the content of the memory block using those elementary types, and possibly a few others. Converting the microprocessor state is, however, not that easy. No direct correspondence, in general, exists between a certain configuration of the internal registers of a certain microprocessor and another configuration valid for another machine. The machine-level code executed, in the two cases, is different, and the program counter, valid in one case, may reflect a position, in the code, that results from the effect of code optimisers, and has no direct correspondence in the code available for the second architecture.

A more detailed discussion about the possible techniques that can be used to implement type tracking and state conversion at every machine instruction for heap blocks, stack contents and machine registers will follow.

2.3 Techniques

There are different techniques that can be used in order to track types as required, depending on whether a more static or more dynamic approach is adopted. In a purely static approach, the compiled, native code is left completely unchanged, and the type information is obtained, when necessary, exclusively by looking at tables or data structures created statically during compilation. A dynamic approach, instead, will discover all, or part, of the needed type information while the program is running, actively monitoring, for instance, the call chain, memory accesses, or other runtime behaviours of the program.

The obvious advantage of an entirely static type discovery is the absence of additional code to be executed, and consequent overhead, at the expense of a greater memory occupation due to the size of all the data structures that need to be prepared in advance. In particular, as previously mentioned, maintaining static type information about all of the program data for every machine instruction can be quite onerous in terms of space occupation. On the other hand, a dynamic analysis is potentially able to obtain much more information about the running code, and would therefore be the only viable solution in certain cases.

As previously mentioned, space occupation is nowadays much less of a concern than it used to be, while using fully optimised code, without runtime overhead, would enable a greater system

efficiency in many respects. Consequently, the analysis presented in this thesis will concentrate on techniques that rely, whenever possible, exclusively on data structures that can be generated statically, using information that can be obtained from the compiler, and that do not require the generated code to be changed or de-optimised in any way. As we shall see, an entirely static approach can actually be used in a surprising variety of cases, covering practically all the needs of usual program code. That paves the way for an implementation of a compiler toolkit in which the code produced is unchanged with respect to the code generated by a plain compiler, and still all of the mentioned services (garbage collection, persistence, migration, etc.) are made transparently available as desired.

It is worth mentioning that all of the issues described here in the case of heterogeneous environments (persistence and migration) refer to executable code obtained automatically for the different platforms from a common high-level representation, for instance the same source code or virtual machine code. Issues like code evolution, or binary translation, will therefore not be treated here directly, although many of the techniques described in this work are certainly relevant, and could be useful in those contexts as well. As far as the actual distribution of the multiple executables on the different machines is concerned, there are several different alternatives that can be adopted (fat or slim binaries [FK97], Just In Time compilation using virtual code [Arm98, SOT+00], etc.), but in general the problem is orthogonal to state capture and migration and can be treated independently, hence it will not be further investigated here. In the remainder of the text, it will be assumed that the thread data that may need to be inspected or manipulated by the service routine is contained in the heap, the stack, and the registers. The global data area can be considered functionally equivalent to a permanent first frame in the stack and, unless explicitly noted, it will be treated accordingly.

2.3.1 Type tracking and data conversion in the heap

Since the system has to keep track of all the memory allocations in the heap, and of the type of every data item present, the most obvious solution is the use of a customised memory manager, which is actually part of the support infrastructure for the various services. In addition to handling the various requests for memory blocks, like any heap manager, the customised memory manager will have to offer facilities to track the types of the data contained in the heap, so that the modules implementing the different services can inspect and alter, if needed, the heap contents relying on that information.

In most high-level languages, the type of every memory block allocated in the heap is known at the moment of its allocation, with a few exceptions that will be extensively discussed later, and every component of an aggregated structure has a well-defined and fixed type for the entire lifetime of the structure. Maintaining the necessary type information is therefore straightforward if the memory manager requires every allocation to be performed specifying, via a descriptor of some sort, the exact type structure of the memory object being allocated. It is usually possible to decide the content of such a descriptor in an entirely static way for all statically typed languages. In dynamically typed languages, even if variables can change their type during execution, each value has a fixed type, known dynamically, according to which the data manipulations are performed.

When a memory allocation is requested, therefore, the type structure of the newly allocated block is known and a suitable descriptor can be obtained.

Recalling the distinction between the two cases, homogeneous vs. heterogeneous environments, the needed descriptors will include information about the location of pointers in the structure being allocated, or, respectively, the complete layout of all the types of the components used, possibly including additional information concerning padding.[1] Some problems may arise when the types of the components of the memory blocks allocated in the heap change during execution, as it may happen when untagged C unions are used. In that case a group of memory locations can be interpreted in multiple ways, depending on the particular instant during the execution and on the program logic. The problem will be discussed in Section 2.4.

Knowing the type of every memory object present in the heap, performing a conversion in a format suitable for persistent storage, or for use on a different architecture, is relatively straightforward. In the first case it suffices to replace the pointers with persistent identifiers, or vice-versa. In the second case, the natural representation in one of the architectures (endianness, padding, etc.) can be easily converted in another representation suitable for a different environment, since there is in any case a simple mapping between the two representations.

2.3.2 Type tracking and data conversion for the stack

The stack contains, during the normal program execution, several data areas juxtaposed, which all grow and shrink according to the way in which the control flow enters and exits the different subroutines. In order to convert the entire stack content when a migration on a different architecture is required, all those different areas should be tracked and converted appropriately. It is therefore useful to analyse the main structures situated on the stack, according to their use.

The most obvious use of the stack is the allocation of the blocks of local variables used in each procedure. Tracking those structures is relatively easy, since the contents of the frame allocated, at least for statically typed languages, are determined by the compiler using a static analysis of the original source code. On the other hand, the local variable area may be used in different ways depending on the position in the code (some memory locations might be shared among multiple local variables), in which case additional descriptors will be needed to keep track of the situation. Similarly, the temporary values area, used while calculating expressions, is maintained on the stack and might change in content and size multiple times during the execution of a single routine, therefore a special analysis, exploring the situation instruction by instruction, is required.

Other critical areas on the stack that need special handling are the register save area, the area reserved for the return value (if present), and the return address. The specific details about suitable techniques to use in order to track the different areas of the stack will be discussed in Chapter 6. As we will see, it is not excessively difficult to devise strategies in order to achieve the intended result.

[1] Since padding is used in a way that is defined by the specific Application Binary Interface in use, the heap manager should actually be able to reconstruct the padding information and the complete physical layout of every structure allocated just by looking at the sequence of the types of the various structure fields. That would clearly require perfect agreement between the padding algorithm as implemented by the heap manager and the compiler, but it is in principle possible.

2.3.3 Type tracking and data conversion for registers and microprocessor state

Keeping track of the values contained in the registers, and their types, is quite a difficult task. During the execution of optimised native code the registers can be used to perform a number of different operations. For instance, the same register could be used in different moments to perform arithmetic computations, as a temporary cache for pointers, or to calculate addresses. In most architectures many registers are general purpose, and can be used indifferently for both scalars and pointers. Since the specific use of each register depends on the exact machine code that is being executed, and can vary from one instruction to the next, keeping tracks of the exact situation is rather complex.

A possible solution, that could allow the system to track the types of the values contained in the registers, is the creation, alongside the executable code, of data structures that map each value of the program counter to a description of the types of the data contained in registers at that point (the "PC maps" mentioned in Section 1.2). If the code is generated automatically by a compiler, the information necessary to build the above mentioned structures is already contained in the compiler structures while the compilation proceeds. Modifying in a suitable way an existing compiler, or designing a new compiler appropriately, it is therefore possible to generate the necessary tables in an automatic way. An extensive discussion about how this result can be obtained, the problems involved, and the technical solutions, will be one the main components of the following chapters.

The data conversion can be performed in a straightforward way if the only kind of transformation required involves modifying pointers, or substituting them with a more abstract representation as a persistent identifier. That is the only significant issue if a single microprocessor architecture is involved. However, if a heterogeneous environment is required, the content of the registers cannot be simply transferred. Different architectures can have different registers and the program counter may refer to code that was optimised using different techniques, leading to operations executed in different orders, and calculating different intermediate values. More generally, the internal state of the microprocessor is highly dependent on the specific characteristics of each architecture. Transforming the microprocessor state into a form suitable for another architecture is quite complex, and would involve applying a non-trivial mapping between each value of the program counter and the registers' content on one architecture and a different value of the program counter and a different registers' content on a target architecture, so that the computation can resume properly.

While the contents of heap and stack can be transformed into the corresponding format of a different architecture, the resulting data could be unusable to restart a proper computation. For instance, as a result of different code optimisations, certain operations on data contained in the stack or the heap could be executed in a different order, and an intermediate state of the memory, if simply translated and transported, could be inconsistent with the operations performed by the code on the target machine. While many of the techniques presented later could be used or adapted to the case of heterogeneous migration, the context assumed for most of the later discussion will be the use of a homogeneous environment.

2.4 Possible problems

Some problems might arise in those languages in which the type of some components of an object can change during its lifetime, for instance using records with variants in Pascal, or C unions. In that case some memory locations can contain values of different types depending on the dynamic behaviour of the program, and sometimes it might be simply impossible to determine the current types without actively tracking every memory access to those locations. It follows that either the code or the memory representation of those structures may have to be changed if unions are to be supported. Tracking dynamically the memory accesses would be rather expensive, and a more convenient solution might be reshuffling the memory representation of those unions so that there is never any ambiguity in determining the type of every datum contained in any memory location. Depending on the intended use of the type information, heterogeneous migration vs. homogeneous migration, that would mean respectively to allow only fields with identical type to share the same locations in memory, or, if a single architecture is involved, not to let fields containing pointers and scalars to overlap with one another, so that pointers and scalars are kept physically segregated in the memory representation of the union record [Bar88]. The obvious downside would be that the offsets of the various members inside the structure can change, which might cause problems if the programmer is allowed to use those implementation-specific aspects to control member aliasing, for instance. Another possible issue is the interoperability with pre-existing data files, containing unions that using the standard layout.

Similar problems can arise in those languages that allow a rather liberal type conversion between scalars and pointers to take place, or allow union types to be used for the same purpose, bypassing the type system, often relying on non-standard or non-portable compiler behaviour. In those cases, values that appear to the compiler as ordinary integers might actually be pointers, but their being outside the system control would then prevent the correct transformations from taking place. The only viable alternative, in those cases, is to insist that such cast conversions (from pointers to integers and vice versa) can never be used in user programs, or alternatively that every direct manipulation of the value of pointers is performed using system utilities, so that the necessary information is preserved. It is worth noting that forbidding that kind of conversions is not actually such a drastic restriction, in typical user programs, and that many modern programming languages, most notably Java, hide completely the implementation of pointers as numeric entities, identifying direct pointer manipulations as a source of considerable programming and debugging problems.

Another detail that might be encountered while attempting to perform a data conversion is the use, sometimes made by compilers or languages, of packed data structures in which integers, bit fields, and other data types, are not kept aligned on a whole byte boundary but are instead on an arbitrary bit boundary. Despite the added complication, performing the conversion is not, in this case, significantly more difficult from a conceptual point of view. It will suffice to keep track of types to the bit boundary, instead of to the byte boundary, in order to have again all of the information needed to correctly perform the necessary unpacking and conversion of the relevant data. For instance, knowing that there is an integer 21 bit wide, in little-endian format, beginning from bit 3 at a certain byte offset in the heap is enough to extract its value and change

its representation into a different format, suitable for a different use.

Another relevant problem is the interaction with the host system. The assumption made until now is that the entire information needed to describe the state of a thread can be found in the heap, on the stack and in the microprocessor registers. This is not always the case, though, if the support for migration, persistence and GC is built on top of a pre-existing operating system that preserves some state in its own data structures. In that case, the state stored in the operating system's internal structures would not be under the control of the customised memory subsystem. It would be therefore impossible to trace their content as necessary, and to extract such state when needed. For example, files and network sockets in use are managed by the host system in a manner that is not necessarily known to the user program. Unfortunately, there is not much that can be done to solve the problem easily. If the host operating system is not designed to allow an accurate inspection of its internal state, there is no easy way to extract the necessary detailed information, which is simply not maintained. A possible workaround could be in that case the implementation of a local cache, under the control of the custom memory handler, for all of the information that must be available on the destination system, for instance in order to reopen the files, recreate the sockets, redraw the screen, and so on.

Finally, keeping track of pointers is easier if the only pointers considered are those pointing to the base location of blocks of memory contained on the heap. If a pointer refers logically to a certain block, but points to a location that is displaced from the base location by a certain offset, maintaining the association between pointer and block of memory can be more difficult. More details on this aspect will be given in Chapter 10, devoted to derived pointers.

Though this be madness, yet there is method in 't.
*— **William Shakespeare (1564 - 1616)**, "Hamlet" act 2 scene 2*

Chapter 3

Implementation Techniques

We have seen so far what must be implemented in order to offer a system capable of supporting services like garbage collection, persistence, migration, etc. to operate preemptively, at every machine instruction, without requiring additional instructions to be inserted in the optimised compiled code. We have discussed why the most crucial requirement is the ability to track types and to perform data conversions for all of the information contained in machine registers, heap, and stack. Subsequently, we have analysed the challenges involved in creating an implementation of the mentioned support.

This chapter introduces some techniques that can be used to address the challenges previously described. A more precise description of the working context is given, and the concept of "mode" is introduced. Particular features of compilers and microprocessors that can introduce additional complexity are discussed. The content of the following chapters is introduced and discussed in general terms. The experimental prototype, described in Chapter 9, is also introduced.

3.1 Existing techniques

One of the earliest projects in which support for preemptive garbage collection was offered is the Trellis/Owl system, as described in 1987 in a paper by Moss and Kohler [MK87]. According to Eliot Moss, the compiler used by that system was not fully optimising, and there might have been some earlier compilers for AI languages (LISP, etc.) that used similar techniques. Much more recently, the Java compiler developed at Intel, described by Stichnoth, Lueh and Cierniak [SLC99], and the MOBY system, by Fisher and Reppy [FR02], are both able to support garbage collection at every assembler instruction on the IA32 system. A port of MOBY to the PowerPC is reported to be currently in progress.

The aforementioned Java compiler, described by Stichnoth et al., was designed with the built-in ability to create PC maps, for every instruction, during compilation. Their maps are constructed, relying on the almost complete type-safety of Java, to discover which stack locations and registers

contain pointers at any given instruction. The only exception to type-safety in the Java bytecode (deriving from the "finally" clause, see Agesen [AD97]) is treated specially.

Other compilation frameworks, in general, have no native ability to generate PC maps, and adapting an existing compiler is usually far from easy. Bernard, Harper and Lee [BHL98], in their work on TIL, report that "Pseudo registers in MLRISC [...] carry no trace values or type information; there are distinct classes of integers [...] but an integer pseudo register that happens to be used as a heap pointer is not distinguished in any way." Their PC maps (for selected points in the code) were built by reconstructing the mapping from final registers and stack locations to the pseudo registers used in the intermediate representation.

The authors of MOBY have used a mapping similar to the one used in TIL to map the abstract representation of registers back to the variables used in the intermediate representation. While PC maps are created for every machine instruction, MOBY does not actually obtain fully accurate pointer information, and relies instead on a mostly-copying garbage collector. Reportedly, MLRISC has added a better support for associating information with registers in recent revisions.

Another example of similar techniques is offered in the context of a Modula-3 compiler, based on GCC, which was adapted by Diwan et al. [DMH92] to support garbage collection using PC maps. The optimising compiler GCC 2.0 was modified so that PC maps are produced to help the garbage collector in finding and updating all the pointers in the stack and in registers. A set of tables was produced for each point in the code in which garbage collection is possible, but not for every machine instruction. The code was generated for the VAX architecture using Ultrix. Diwan stated that the modifications to GCC were both in the back end and in the language front-end, which was customized to pass additional information to the back end. Notably, the system offered support to pointers that might not refer to the base location of a memory block (see Chapter 10). By comparison, the aforementioned Java compiler described by Stichnoth et al. avoids all optimisations that generate derived pointers. A paper by Shivers et al. [SCM99] describes an interesting approach to atomic allocations, and the interconnections between garbage collection, interlocking, atomicity, and preemption, in the context of a customised implementation of SML/NJ.

During my internship in Sun Microsystems Laboratories, I also had the opportunity to discuss in detail with Ross Knippel, David Cox and Chuck Rasbold the design of the JBE Java compiler. Derived from the UBE (Universal Back End) compiler, JBE builds GC-maps to determine the location of pointers in registers and stack at specific points in the code although, according to their description, the map generation could have been done, in principle, for every machine instruction. Unfortunately, the compiler was developed strictly as a production tool, and no publications were made available about the project. However, a summary of the system details relevant to the present research is reported in Appendix A. JBE eventually became the optimising compiler included in Sun's ExactVM, together with the basic Java interpreter and a simplified JIT compiler. Sun's ExactVM is also known as EVM, as ResearchVM, and as JDK 1.2 Solaris Production Release.

Leaving aside garbage collection, there seem to be no examples of systems in which migration or checkpointing of individual threads can take place at arbitrary machine instructions, with the ability to move memory blocks. There are, however, systems in which the state of the whole system is saved preemptively [SSF99], or single threads are migrated preemptively, but without

allowing any structure to change address in memory [AP99, ABN99]. The explicit recording of all the pointers that would need to be updated, with the obvious penalties in terms of performance, has also been suggested [CHM97] as a (crude) way to enable the movement of memory blocks in the heap while operating preemptively.

The compression of PC maps, or similar structures, has been discussed separately by Diwan [DMH92], Tarditi [Tar00], and Stichnoth [SLC99]. Those works show how the necessary tables can be compressed, in many cases, quite efficiently, which further reduces the concerns about the space occupation of the maps used to implement preemptive services. A useful paper by Boehm and Chase makes important considerations on the effect of code optimisation on derived pointers [BC92].

The listed references show how PC maps can be obtained from the knowledge that the compiler has about the location of pointers. However, these all refer to very specific environments, languages and target architectures. When a generic infrastructure has been used, the changes applied to generate PC maps were not limited to the back end, making it difficult to adapt the system to different high-level languages. In the available literature, the main emphasis has been on preemptive garbage collection, but no consideration has been given to the wider range of applications that would be available using that approach. The `C--` system [JRR99, JR98] defines a portable back end and a runtime system, and defines an interface that can be used to supports multiple program services. The separation used by that system between runtime core and runtime services, and the definition of a clear interface between them, is similar to the approach suggested in this work. Threads in `C--` can only be suspended at a safe point, however, so there is no support for preemptive services.

The following discussion will try to adopt a neutral approach with respect to the specific services in use. Furthermore, the analysis will be conducted from the point of view of the back end, trying to minimise the overall dependencies from the rest of the compiler. If the PC maps could be obtained by customising only the back end, the whole system would be usable with multiple front ends, and therefore with different high-level languages. A given set of specifications for the source programs (the use of the customised memory manager, for instance) would then assure compliance with the system requirements, and all the code compiled with the modified compiler would automatically acquire the ability to have garbage collection, persistence, etc. Even better, custom services could be plugged into the system in a seamless way. The possibilities certainly look very interesting. The prototype described in Chapter 9 is indeed able to generate automatically PC maps from source code written in multiple languages. Further references, more specifically related to the customised liveness analysis described in Chapter 5, are listed at the end of Section 5.1.

3.2 The context

Tracking pointers, in general, can be a rather challenging task, especially if a relative degree of independence from the specific high-level language is desired. Many programming languages do not handle pointers in a very abstract way, rather considering the numeric value of the pointer

as all the required information. For instance, the C language allows the programmer to convert freely a pointer into a scalar value and vice versa. The compiler, however, will be unable to detect pointers that are manipulated, in parts of the program, as simple integer values.

Similarly, it can be problematic in certain cases to discover the connection between a pointer and the memory block to which it logically refers. In certain cases a pointer that is used as a base pointer for certain memory operations can even point outside the address range to which it refers (for instance), as discussed in Section 3.10 and more extensively in Chapter 10. Furthermore, in a compiled program several kinds of pointers can be present: pointers to the code, pointers to the stack, pointers to the heap, pointers to system structures and so on.

While those problems can, in general, be addressed in various ways depending on the specific situation and the functionalities required for the particular memory management operations involved, trying to fully support languages like C and C++, with their low-level handling of pointers, would prove to be a particularly frustrating, and probably pointless exercise, as suggested by the existing efforts to add some form of automatic garbage collection to C and derivatives. To be able to explore the problems involved in the tracking of pointers while maintaining a more general view of the problems, it is therefore useful to define more clearly the typical environment and language features that we want to be able to support. The result will be a set of guidelines to which the high-level source code should conform in order to gain automatic pointer tracking, and an indication of the techniques that can be used by the compiler to offer such a feature.

While the exact details of what can be allowed in the source code will become more evident throughout the discussion of the implementation techniques, the context can be summarised as follows. First of all, the system is assumed to support a family of imperative high-level programming languages that make use of a stack and a dynamic heap to store their data. While all the pointers will be considered, special attention will be paid to pointers which refer to heap blocks. It will also be assumed initially that those pointers refer to the base location of heap blocks, and the more general case (derived pointers) will be discussed in Chapter 10. It will be necessary to require that all the pointers can be recognised by the compiler. Consequently, conversions of pointers into scalars and vice versa will have to be disallowed. Similarly, unions in which pointers can share memory with scalar values will have to be treated with care. Absolute pointers, not related to structures that can be relocated (code, stack, heap) may create problems, and their use is not advisable. Finally, it will be necessary to specify the layout of heap blocks, so that their pointers can be found at runtime.

Despite the fact that those restrictions might appear stringent, most of the “well-behaved” programs written in modern programming languages will actually conform to the specifications quite easily. For instance, in a Java program the objects are allocated specifying their class (from which the object layout can be easily obtained). Java pointers are handled implicitly by the language and are kept distinct from scalar values. It is very common for an implementation of Java pointers to have them always referring to the beginning of an object. Consequently, ordinary Java programs, compiled to native code, could be supported with relative ease.

3.3 A test environment

The test environment used to verify the feasibility of such an implementation was GCC, version 3.3.3 [Stab]. Only modifications to the back end have been used, with good results. The test architecture used for the prototype was the SPARC V8 [Spa92], although similar modifications could be easily applied to back ends of different microprocessors as well.

3.3.1 Why GCC?

As previously mentioned, the idea behind the tables generation was to extract information about the pointers, and more generally about the primitive types used in the various data structures, directly from the compiler. Furthermore, the modifications should ideally be confined only to the back-end, and no change to the front-ends should be needed. The ideal candidate for the test environment should have a clear defined separation between the front-end and the back-end, and the source code of the compiler should be readily available.

Various compiler suites were considered (some are mentioned further in the text), but GCC was found to have the following advantages:

- It is modular and easily customisable.
- It is open source, therefore all the source files are widely available without charge.
- It is an industrial-strength product, extensively used world-wide.
- It is constantly maintained and updated.
- Several front-ends are available for many of the most popular languages (Ada, Java, C, C++, FORTRAN, COBOL...), which allowed a more extensive experimentation using different front-ends.
- It has been ported to a large number of different architectures, and its inner core is certainly flexible enough to support any microprocessor architecture commonly available. The standard distribution supports some thirty different microprocessors (i386, SPARC, MIPS, MC680x0, Alpha, ARM, VAX, PowerPC, SH3, etc.), and more are supported using additional packages.
- The interface with the back-end is clearly defined and has been stable for many years. The back end description for every architecture is contained in a small number of files (typically just three). Everyone can easily create new back ends, or customise existing ones, following the available documentation.

Being such a widely used tool, implementing the desired support in the context of GCC shows how the techniques described can be applied to a real development environment. Part of the analysis developed in the following chapters, in particular the content of Chapter 5, is driven by the desire to simplify the implementation of PC maps in the context of an existing compiler, rather

than re-design everything from scratch. Among the other compilers considered, LCC [Fra91] could have been a possible choice, but its being exclusively a C compiler would have prevented experimentation using multiple front-ends. SUIF [ADH+00] was another possible choice, but the environment was not fully stable at the time. Both SUIF and SUIF2 have not seen new releases in several years, now. Other compiler suites considered either were closed-source, required payment of fees to access the source (for example, the Amsterdam Compiler Toolkit [TSKS83]) or were limited to single languages or single microprocessor architectures.

Overall, considering all the points mentioned above, GCC seemed to be the most appropriate test platform. A minor drawback of GCC is its relative internal complexity, but in this case only the back end needed to be analysed and customised. The results eventually obtained seem to confirm the suitability of the choice.

3.3.2 SPARC v8

Among the many different microprocessor architectures supported by GCC, it was necessary to select a first test case to proceed with the implementation. Since part of the problem is tracking pointers in registers, the architecture chosen should have several registers. The availability of the architecture was also a factor taken into account. Among the possible candidates were Alpha, PowerPC and SPARC. The SPARC v8 architecture was eventually chosen, mainly to verify whether even an architecture with some uncommon features (window registers and delay slots, among others) can be successfully supported. Any other architecture could have been selected for the test, and further tests might be conducted in the future using different microprocessor families.

3.4 Types and modes

3.4.1 Modes

The concept of type, as it is used in high-level languages, is fairly complex. Different notions of type equivalence (by name, structural,...) combine with a variety of type constructors, primitive types and other aspects in sometimes very sophisticated and complex type systems. As previously discussed, the level of abstraction needed in order to manipulate memory is much lower, and it is the minimal required in order to perform the necessary translations and conversions.

If a single microprocessor architecture is in use, it is sufficient to know whether a register, or a memory location, contains a pointer or a scalar value. If multiple architectures are involved, conversely, or if complex manipulations are required on the internal data representation, it might be useful to know which of the different primitive types is assigned by the compiler, at any machine instruction, to registers and memory locations. The whole complexity of the high-level type system is not necessary. Furthermore, as explained later, there are cases in which the registers or memory locations contain temporary values that result from operations performed at the machine language level, and that have no exact correspondence with any type used at high level.

To avoid possible confusions, the low-level type interpretation of the content of registers and memory cells will be called, from now on, “mode.” Therefore we will say, for instance, that at a

certain point in the code the mode of a register is pointer, or scalar. Or, if we need greater detail, we may say that the mode is 8-bit scalar, or 16-bit scalar, or pointer and so on. The mode is, in general, loosely related to the way in which the compiler uses registers and memory to reflect the primitive types used in the high-level program code. However, in some cases, they may derive in non-trivial ways from temporary values or from particular expansions of high-level constructs.

As previously mentioned, the most important distinction among the possible modes is certainly the one between pointers and scalar values. However, depending on the kind of manipulations that are required on the data, or possibly on the code, more refined distinctions are possible. It may be useful to define distinct modes, for instance, for the following:

- pointer to a heap block
- pointer to a stack location
- pointer to executable code
- integers of various sizes
- floating points of different sizes
- unused
- “split pointer”

Knowing that a register contains a pointer to the heap or to executable code can simplify, in certain cases, its handling. For instance, if we are interested only in heap manipulations, all the pointers to executable code can be safely ignored, avoiding the need to check at runtime for every pointer whether it really refers to a location inside the heap or not. Knowing that a register is unused at a certain point can be used to avoid unnecessary operations. For instance, it is not necessary to perform data conversions on the content of registers and memory locations that are known to be unused. It is worth pointing out that a specialised floating point register can indeed have a mode different from “floating point”. For instance, a floating point register might be actually used to store integer values or even pointers, as a buffer for other registers, or it might be simply unused. The “split pointer” mode will be discussed in detail in the next section.

Ideally, the compiler should be able to distinguish internally all of the listed modes, so to allow us to produce tables containing the most accurate information. However, such a fine distinction is not always present in the low-level intermediate representation, available before the machine code generation. The tables will have to be built, therefore, with the kind of information that is available. For instance there might be no indication of whether a certain register is unused while some code is generated. In that case, it might be necessary to inspect the intermediate representation to reconstruct the lifetime of the register’s content. Another example could be the lack of distinction among multiple kinds of pointers. If all the pointers are treated in the same way, it is then necessary to check them at runtime to distinguish pointers to the heap from pointers to the stack, and so on.

The fact that all the modes listed above refer to logically different, albeit low-level, uses of values, can be taken as an indication that compiler designers should really introduce more categories of primitive data in the low-level intermediate representation used by the compiler. GCC, as we shall see, offers a fairly extensive range of low-level categories of values, which can be remapped onto modes in a way that is fairly adequate for our needs. Some postprocessing of the information extracted from the compiler will be applied on the information extracted from the back end, as explained in Chapters 5 and 10.

3.4.2 Split pointers

A peculiar condition may arise in RISC machines, in which all the machine code instructions are of the same size and there is not enough space inside a single word to fit both the opcode and a long literal. In that case, RISC microprocessors usually offer two distinct machine instructions, used to load respectively the high and the low part of the literal in a register. For example, let us force the compiler to return a 32-bit value as a pointer, so that we can compare the result on different microprocessors.

The original C program is:

```
char *xyz() { return(char*)0x6abc92fd; }
```

The code fragment is expanded as follows:

Microprocessor	Expansion
x86	`movl $1790743293, %eax`
680x0	`move.l #1790743293,%a0`
PowerPC	`lis 3,0x6abc` `ori 3,3,37629`
SPARC v8	`sethi %hi(1790742528), %o0` `or %o0, 765, %o0`
MIPS	`li $2,1790705664` `ori $2,$2,0x92fd`

Table 3.1: Loading a 32-bit constant in a register

As shown in Table 3.1, while CISC machines (x86, 680x0) offer instructions to load the entire 32-bit literal in one step, RISC machines (PowerPC, SPARC, MIPS) must load separately a high and a low part. In particular, MIPS and PowerPC load two 16-bit half-words, while the SPARC v8 loads initially the 22 high bits, and then fills in the remaining 10 bits using the "or" instruction [Spa92].

That causes an interesting problem if the program is preemptively stopped exactly in the middle of the two-step sequence. If the value being loaded is a pointer, while the previous content of the register was a scalar, for instance, the partially loaded register is not yet a meaningful pointer, and yet it is no longer a scalar. The mode of such a register at that point will be defined

as “split pointer”, to denote its peculiar state. A split pointer pointing to a heap block cannot be modified as easily as other values, since its low part will only be loaded when execution is resumed.

Potentially, handling such a value could be problematic, since at runtime it is not obvious to which heap location the split pointer will eventually refer once complete, and consequently what adjustment should be applied in case some heap memory should be moved. In practice, however, the problem does not normally arise. First of all, loading a literal as a pointer involves the use of absolute pointers. User code, with the exception of low-level code, does not usually contain absolute pointers, as they are discouraged by modern programming practices. Furthermore, programs which allocate memory on the heap will only obtain back from the heap manager complete pointers and there would be no reason for the code to deal with absolute references.

Depending on the implementation, the compiler might generate absolute pointers when referring to global data. Those pointers, however, do not refer to heap objects, and are therefore not involved when a heap reorganisation is necessary. Similarly, if absolute pointers are used to refer to memory-mapped I/O devices, their values do not need to be adjusted when the heap is reorganised. If the heap is the only area in which memory can be moved, therefore, split pointers can be left untouched as long as the user code never addresses the heap using absolute pointers, which is by no means a stringent restriction.

If the manipulations required, on the other hand, require globals, code, or stack to be moved and absolute pointers are involved, a bit more work is required. To begin with, it would be quite unlikely to encounter absolute pointers referring to a location within the stack. Relocating the stack does not pose particular problems, therefore, as far as split pointers are concerned. Absolute pointers to code locations could be present if absolute code is generated by the compiler. Those occurrences can be easily eliminated if the compiler is able to generate position independent code, which is usually the case on many environments. Similarly, globals could be accessed using an absolute address, but they will not be present if the compiler can be asked to generate instead accesses referred to a certain base register in order to access globals.

If absolute pointers are really desired, it is still possible to relocate globals and code if they are moved as a single block and it is possible to distinguish split pointers that refer to those areas from split pointers pointing somewhere else. For example, on the SPARC a split pointer differs from a complete pointer only in the ten low-order bits. That means that a split pointer can be at most 1024 bytes displaced with respect to the corresponding complete pointer. If globals, stack, code, heap, and other data present in memory are all separated by at least 1024 bytes, it is always possible to tell to which area a certain split pointer refers to, and adjust its value accordingly if one of the areas is moved.

If still more control is desired, finally, it is possible to adopt yet more solutions. For instance, if the two instructions that load the high and low part of the pointer are contiguous, the short sequence could be treated as a sort of critical section. Alternatively, a flag could be added to the PC maps so that the runtime module can recognise that the interrupt happened between the two. Upon return from the service routine, the two-instruction sequence can be re-executed from the beginning so that the pointer is reloaded correctly.

In fully optimised code, however, it is not infrequent to treat the load-high and the load-low

instructions independently and, because of code optimisation, the two parts could be separated by an arbitrary amount of code. In that case, it is still possible to preserve in the PC maps the association between the register containing the split pointer and the expression used to compute the address that will be eventually loaded. When a service routine is called, and some memory is moved, it is then possible to recalculate the full address and reload the value of the split pointer accordingly. Chapter 10 discusses this solution in more detail.

3.5 More elements to consider

The base concept of PC maps is relatively simple, but there are some details that must be taken into account. For example, it is necessary to deal appropriately with subroutine calls. From the point of view of the assembler code a subroutine call is just one instruction in a sequence of other instructions, and the PC maps will just specify the state of registers before and after its execution. However, the impact of the instruction on the state of the registers can be rather complex — some registers will be used by the subroutine, others will be overwritten with local values by the callee, and others may be used for return values. It is therefore necessary to determine how the subroutine call, considered as a single entity, affects the mode of the various registers and to calculate the mode maps accordingly. Although the identity of the subroutine which will be called may not be statically known, the compiler knows, for each call site, the list of arguments used and the return value, and the mode used by each of them. The standard specified for a certain microprocessor by the Application Binary Interface can then be used to reconstruct which registers are used for which argument, which registers are preserved by the call, which ones are overwritten, and the general overall impact of the call.

Another important aspect is that the mode analysis described is only able to determine the mode of registers which are actively used within the routine body, but nothing can be said, just by looking at the current routine, for those registers that are set in a certain mode by the caller and are preserved as they were during the entire routine. The simplest way to determine the mode of those registers is just to extract the address of the caller from the stack and look in the corresponding mode table, traversing if necessary the call chain until the mode of all the registers is known. Potentially, the procedure would involve an unknown number of steps, since the depth of the dynamic call chain is not known. In practice, the number of steps required to complete the whole map will be fairly limited, since the typical stack depth in a non-recursive program is generally limited. Besides, the whole stack needs to be scanned in any case in order to discover the pointers contained in each frame. In any case, if an absolute real-time constraint for this particular operation is required, it is possible to adopt a slightly different solution that allows the system to determine the mode of all the registers in constant time, at the expense of a small overhead during calls. By passing from routine to routine the mode mask, for instance by pushing the previous one on the stack and combining it with the map of the callee, the complete mode information is always ready for use, and it can be obtained in a constant number of instructions.

Another interesting problem concerns the prologue and the epilogue of each routine, that is the portions of code appended before and after each routine body. The purpose of the prologue is

to create a new local environment for the execution of the callee (possibly saving some registers on the stack, and creating a new stack frame), while the epilogue should remove the current stack frame, restore the saved registers and transfer the return values where appropriate. In the prologue, for instance, the mode of a certain register might depend on the PC maps defined for either the current routine or the caller, depending on the position in the code. Before the point in the code in which the value of a register is saved on the stack, its mode depends on the caller, while after the value is saved the mode depends on the local map. If multiple push operations are performed, it is necessary to determine with precision whether each register is "owned", at each instruction, by the caller or by the callee. Some additional data will have to be included in the maps, in order to enable the routines called preemptively to determine the mode of each registers. Similar considerations apply, symmetrically, for the epilogue.

3.6 Unusual features of compilers

Optimising compiler systems may sometimes use techniques that can make determining the mode for the various data more complex. One of the features that can hamper the task is the ability to fit, if required, multiple values of small data types inside a single register or block of registers. That may happen, for example, in the "packed record" type in Pascal, or using bitfields in ANSI C. As an example, this is a fragment of C code that uses bitfields:

```
void oo(register int s)
{
 register struct {
  int r:5;   /* 5 bits */
  int c:11;  /* 16 bits used so far */
  int *u;    /* 32-bit pointer, not aligned */
  int e:16;  /* another 16 bits */
 } x;

 x.e=s;
 x.c=s;
 x.r=s;
 x.u=(int*)0x12345678;
}
```

The resulting expansion of the function body, compiled by GCC for the Motorola 68020 microprocessor [Mot92] is shown below (commented for clarity). Note that, in MC68020 bitfields, bit 0 is the most significant bit. If a short integer is stored in a 32-bit register, therefore, the bits used are 16..31.

```
move.w %d0,%d2          % bits 16..31 of d2 (x.e) are filled with s.
bfins %d0,%d1{#5:#11}   % bits 5..15 of d1 (x.c)\  Bits 0..15 of d1
bfins %d0,%d1{#0:#5}    % bits 0..4  of d1 (x.r)/  are used
clr.w %d1
```

```
or.w #4660,%d1          % half of the ptr in d1, bits 16..31
and.l #65535,%d2
or.l #1450704896,%d2    % half of the ptr in d2, bits 0..15
```

As the example shows, the two registers d1 and d2 are used together as a combined representation of the packed structure and neither of them fully contains either a scalar or a pointer, forcing a slight rethinking of the mode description strategy. Although the problem may appear rather complex, there are several possible solutions. A simple strategy is disabling the packing mechanism of the compiler, so that the resulting structures maintain pointers and scalars distinct. That solution, although simplistic, is satisfying in a number of cases, but it might have an adverse effect if the user program needs bitfields to access I/O devices, for instance, or makes use of packed structures created by pre-existing code.

If support for packed structures is required, a more complex but more complete solution can be implemented associating modes not just with separate registers and memory locations, but also with strings, arbitrarily long, of bits. In that case, we would describe, for instance, that in the register block “d0,d1,d2,d3..” we have a pointer from bit n to bit m, while the rest of the block contains scalar values. The information could be extracted from the internal structures of the compiler that describe the packed structure and its association with part of the bank of registers, or with a certain area of memory. The tables required to maintain this more precise information would be somewhat more complex, but nothing would change from a conceptual point of view, since we would still have separate data components, each one with a distinct and well defined mode.

A similar problem may appear, on some architectures, when passing structures by value as arguments, which are then split in multiple registers for the argument passing. Similar solutions can be adopted in this case as well, for instance by forcing the compiler to use memory-based storage instead, or simply by calculating the modes for individual bitfields as previously detailed.

3.7 Unusual features used in microprocessors

Dealing with certain features of some microprocessors can be quite complex. We have already discussed the problems deriving from split pointers. Other peculiarities, sometimes found in microprocessor architectures, can also make the task of determining the modes more difficult.

3.7.1 Register windows

One of such features is the use of register windows. The term refers to the availability of a certain number of physical registers in the microprocessor, only a subset of which are visible at any given time to the user program. At least in principle, just by sliding the window of the available registers, it should be possible to avoid many of the traditional save/restore of registers on the stack that are typically found in the prologue and epilogue of compiled functions. Consider, for instance, the diagram in Figure 3.7.1.

When the register window is moved, registers R5 and R6 become accessible as registers R1 and R2, while new registers are available, all without accesses to memory and in a single step. Despite the apparent attractiveness of the solution, register windows are generally considered not to be very effective in modern multitasking computer systems, mainly due to the fact that, each time a context switch is necessary, the whole set of used physical registers needs to be flushed to memory and the previous set to be restored. Register windows have mostly disappeared from modern microprocessors, but are sometimes retained for compatibility, notably in the SPARC architecture. Notably, the Itanium processor (IA-64) uses a similar feature, named "register frames" [Pie01].

When determining the mode of registers, when register windows are used, two orders of problems appear. First of all, some values "disappear" in the hidden registers inside the microprocessor, and it might be difficult to determine their mode when they are extracted. Secondly, registers can change name, appear and disappear in a single step, and it is therefore necessary to take into account the necessary adjustments. The first problem is only apparent, since the register windows mechanism is always used in parallel with some backing space in the stack frames. When a context switch takes place, for instance, the operating system takes care of extracting the hidden portion of the physical registers bank and saving the various parts in the reserved space inside the stack frames. After this operation has been performed, all the "invisible" registers are saved to memory and the situation can be handled as if the register windows were never used.

Slightly more complex, but not by much, is considering the effect of the window shifting during the prologue and the epilogue. A single pair of instructions (save/restore on the SPARC) is used to change the current window, with the effect of causing an automatic save of the registers used by the caller and at the same time "renaming" the registers used as parameters for the callee. The save operation can be seen as equivalent to a save on the stack, and the only necessary precaution is to change the attribution of the modes to registers at the same time in which the window shift takes place. The complete compiled routine will therefore have to be divided into three sections: the first, before the save, and the third, after the restore, will use the register window used by the caller, while the code in between will use a different window, and a different mapping of register names over the physical registers.

3.7.2 Delay slots

The term "delay slot" refers to a mechanism (delayed control transfer) by which, when a jump instruction is encountered, one additional instruction, immediately following the jump instruction,

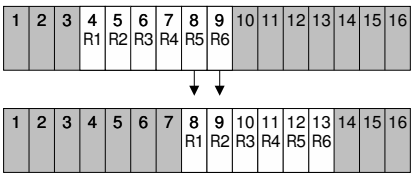


Figure 3.7.1: Register windows

is executed before control is transferred to the jump target. The delay slot itself is the position in the code which contains the additional instruction involved in the delayed execution of the jump. Originally, delay slots were a way to improve the efficiency of pipelined execution, but are now mostly retained for compatibility reasons, notably in current SPARC microprocessors. Similar features, sometimes involving multiple delay slots or different kinds of delayed instructions, were also introduced in the IBM 801, AMD 29000, MIPS R2000 and many others [mWHC92].

The reason why delay slots can complicate the construction of PC maps can be understood by analysing the following code fragment:

```
        mov 6, %o1
        b .away
        mov 9, %o0   # executed before the branch
.here:  st %o0, [%fp-12]
```

In this example, there is absolutely no relationship between the occurrences of %o0, despite the fact that the two instructions appear to be consecutive. That code is really equivalent to:

```
        mov 6, %o1
        mov 9, %o0
        b .away
        nop
.here:  st %o0, [%fp-12]
```

in which it is clearer how the two instructions are actually unrelated. The delayed effect of the jump must, of course, be taken into account when reconstructing the life of every register during the liveness analysis. Furthermore, the instruction in the delay slot, under certain circumstances, can be “annulled”. The annul bit, available for certain branch instructions, is a further form of code optimisation that influences the order of execution, and may be used to reduce the number of jumps in tight if-then-else sequences by annulling the instruction that implements one of the two alternatives. For, instance, the code:

```
if (x>4) q=1; else s=2;
```

is compiled by GCC, using the maximum level of optimisation, into:

```
     cmp %o0, 4
     ble,a .LL3  # branch if less or equal
     mov 2, %o2  # -> delay slot
     mov 1, %o1
.LL3:
```

If the branch is taken, the instruction in the delay slot is executed and the execution continues at the label LL3. If the branch is not taken, instead, the instruction in the delay slot is fetched but ignored, and the following one is executed normally.

Even more confusing is the situation when subroutines are involved, since the return address is actually the instruction *after* the instruction in the delay slot. While building PC maps describing the content of registers for values of the program counter in the current function, therefore, it will appear as if the effect on the registers of the call instruction is combined with the effect of the instruction in the delay slot. For instance, consider the following fragment of SPARC machine code.

```
call abc, 0
mov  %i1, %o0
ld   [%o0], %g1
```

If the microprocessor is interrupted right after the call, no effect from the execution of the call will be visible on the registers, since the call has not been really executed yet. Conversely, if the microprocessor is interrupted after the following instruction, it means that the instruction in the delay slot *and* the call have both been already executed. The liveness algorithm must take into account that combined effect in order to create accurate PC maps.

Additionally, there is the possibility that the instruction in the delay slot is the target of an external jump, independent of the call. In that case, a static analysis of the liveness of registers would need to take into account two possible apparent effects of the same instruction, one including the effect of the call and one without. Determining dynamically which of the two possible alternatives is followed can be particularly complex. On the SPARC, a special internal register (nPC) involved in the execution of delayed branches can be inspected during the interrupt to find out which execution path will be followed. Apparently, GCC never uses instructions in delay slots as targets of branches, which partially simplifies the problem.

Due to the effect of delay slots, determining the real content of registers is not as easy for the SPARC as it might be for other architectures. Suitable algorithms were developed in the context of this research by carefully analysing the possible effects of the delay slots in the various cases, and rearranging the internal representation of instructions and control flow. A formal description of the technique is available in Chapter 5.

3.8 Tracking modes in the stack

Nearly all of the previous discussion was devoted to the techniques that are necessary to track the modes of the data contained in the registers. Several problems, however, are also involved in tracking the modes of the various elements present in the stack. In this section we will introduce in general terms the problems related to discovering pointers in the stack. A more exhaustive discussion will be made in Chapter 6.

3.8.1 Stack components

The stack is generally used for all the information that needs to be stored when a new subroutine is called and automatically disposed when returning to the caller. For instance, all the parameters that outnumber the available registers are typically stored on the stack. The save area for

registers that need to be preserved across the subroutine is also contained in the stack, as are the return address (dynamic chain), possibly some space for the return value, the local variables, and the temporary values used when composing functions or performing mathematical calculations. Conceptually, the stack could also be used to store the static chain, used by languages with nested procedures (like Pascal), necessary to find the variables belonging to enclosing procedures.

Many object-oriented languages allocate their objects in the heap, but it is in principle possible to allocate objects which should by entirely dynamic on the stack, if it is possible to prove that those objects will never be used after the end of the current subroutine [HNB99, GS00, GS98]. Such techniques speed up the code by avoiding the overhead required by a heap allocation, but on the other hand require special attention while operations on the heap, a garbage collection for instance, are performed. As far as pointer discovery is concerned, handling such specially allocated objects does not require greater effort than needed for heap objects, as will be explained later. Stack allocation is not very frequently used in practice, mainly because of the additionally complexity in the implementation.

3.8.2 Problems and solutions

In order to discover all the pointers, it is necessary to establish the modes of the data contained in all the components listed above, so that the proper modifications can be applied whenever a memory manipulation is required. Let us discuss briefly the various components that can be found on the stack. The modes of the locations in the registers save area can be discovered since each location is uniquely associated with a register, as it was used by the caller. The modes of the locations used for the arguments that do not fit the available registers are trivial to determine, inspecting the list of arguments which is being compiled, and similarly for the return value. The temporary values area can be treated in a manner similar to the local variable area.

3.8.2.1 Uninitialised pointers

A more subtle issue, however, might arise from those locations which are supposed to contain pointers but are really unused. For instance, a given stack location might be reserved to store a certain local variable, known to contain pointers, but in the first part of the code the location might not be used. Before the first initialisation of the memory location is made, the value would appear to be a pointer, but it would contain some random data left from the previous use of that memory location. The low-level mode that should be really assigned to that location during the first part of the code, therefore, should be “unused”, rather than pointer.

If the location is unused, modifying its content would be harmless (the value is unused anyway), but interpreting the value as a valid pointer during a garbage collection could prevent some parts of the heap from being reclaimed. Such an approach would be, to an extent, conservative, since it would not be possible to determine the exact set of pointers active at any given moment. In order to avoid the possible problems related to previous values left in memory, one possibility could be to initialise the stack slots containing pointers to a conventional value (null, for instance) at the beginning of the procedure. At the expense of some overhead, that would ensure that every value

contained in such a pointer is a valid pointer to an object (even if, strictly speaking, it is possibly no longer really in use).

If it is possible to determine statically the first and the last points in the code when a certain local variable is used, on the other hand, it becomes possible to discover when a stack slot containing a pointer is not really used, and a greater precision in the determination of the pointers set can be achieved. In general, it is not possible to discover the real last use of a pointer unless a dynamic tracing is performed. For instance, if there is a loop in which the same pointer is first read and subsequently written, and the pointer is no longer used after the loop, we can only say statically that the pointer might be alive towards the end of the loop, since it might be reused in the following iteration.

3.8.2.2 Arrays of uninitialised pointers

Determining the mode of single local variables, performing a liveness analysis, is relatively easy. Things, however, become more complicated in the case of arrays. Array elements can be individually initialised and used in an absolutely arbitrary order, using an indexing operation in which the value of the index can be an arbitrary expression. The value of such an index cannot, in general, be statically predetermined. In other words, it becomes impossible to determine statically the first and the last use of each individual array element, and consequently to determine, in the case of an array of references, when an element is a real pointer and when it is unused.

If dynamic tracking is not used, a conservative approach is the only reasonable approach. Similarly to what was described in the previous subsection, it might be useful to pre-initialise all the pointers contained in stack arrays (and pointer arrays in the heap as well, since the problem is similar) to a conventional, “safe” value. Java, in this sense, has the added advantage that every heap object and stack frame is explicitly initialised every time. As this behaviour is not part of the specification of other languages, like C and C++, initialising all pointers in the stack and the heap to a conventional value can represent a significant source of overhead with respect to the standard compiled code.

3.9 Tracking modes in the heap

Determining the modes of the various parts of memory blocks in the heap is probably the easiest part. As mentioned, we are assuming that all memory allocations take place through a custom memory manager, which is integrated with the memory manipulation subsystem. If we require each memory allocation to specify, using a descriptor, the structure of the record/memory block/object which is being allocated, it is quite simple to detect the modes (at the very least the pointer/scalar condition) of all the data inside the allocated block.

A possible problem might arise from the use of unions, or, in Pascal terminology, variant records. The overlay among the different variants may easily cause scalars and pointers to share the same physical storage. In principle, it is possible to reorganize the record structure so that no scalar ever shares a memory location with a pointer. This solution might break binary compatibility with code generated by a pre-existing compiler, but would allow some form of support for

variant records. As a side note, it is useful to consider that, in modern object-oriented languages, variant records have been effectively replaced by objects, using inheritance to extend in different ways a basic structure, without the inherent problems that variant records have.

3.10 Pointers and derived pointers

Until now we have discussed extensively the handling of scalar values and pointers. The implicit assumption was that pointers refer, when pointing inside the heap, directly to a memory block allocated using the facilities offered by the custom memory manager. In certain cases, however, pointers might not refer directly to the base location of such blocks, but more generally might refer to a certain memory block even without pointing to it. The obvious case is the pointer arithmetic available in C, where a pointer can be made to move inside an array to access one element at a time.

The use of virtual origins for arrays is another example. Consider for instance the following Pascal fragment:

```
VAR a:ARRAY [4..9] OF INTEGER;
BEGIN
 i:=a[j];
```

If certain optimisations are applied, the pointer used to refer to the array could point not to the beginning of the array, but to the position that a hypothetical element `a[0]` would have, so that the indexed access can retrieve the correct element without requiring adjustments to the index. This form of access implies that the value contained in the pointer is obtained from the base pointer by adding to it, possibly multiple times, certain offsets. A similar form is used when, for instance, a component of a record is passed to a procedure whose parameter is a reference to a variable (forms VAR in Pascal, and T& in C++). In both cases, the logical link between the original block of memory and the derived pointer must be preserved even if the memory block to which the pointer refers is moved.

The handling of this kind of pointers will be discussed extensively in Chapter 10, but the main choices are either preserving the exact expression that leads to the derived pointer, so that the pointer can be recalculated if some memory is moved, or alternatively trying to discover the memory block associated with the derived pointer by evaluating the possible values that the derived pointer might assume at runtime.

Another issue related to the handling of pointers is the handling of arbitrary conversions between pointers and scalars, as may be done in C. Such conversions may cause pointers to be unrecognisable as such by the compiler, and would interfere with the mechanisms of pointer discovery. It is worth pointing out that such conversions are typically only used while writing low-level code, and can otherwise be avoided in most cases.

"Have a nice day!"
"Of course I will!... but when?"
— Anonymous

Chapter 4

Pointer Discovery in the Registers

In the previous chapters the idea of PC maps was introduced, and some of the problems that are involved in their creation were briefly discussed. In this and in the following chapters, a much more in-depth analysis will be conducted on the techniques that can be used to discover the pointers present in the system while using PC maps. This chapter, in particular, focuses on the discovery of pointers in registers at every machine instruction. A particular form of liveness analysis, useful to this end, will be discussed in detail in the next chapter. Chapter 6 and Chapter 7 will describe, respectively, pointer discovery in the stack and the heap.

4.1 Introduction

As previously mentioned, determining whether registers contain pointers or not while running compiled code can be rather complex. The main factor that makes the task difficult is the fact that registers can change their mode very often, potentially at every machine instruction. Creating PC maps detailed enough to support preemptive pointer discovery, therefore, requires the mode information to be available with the level of detail of the single machine instruction. Adopting a purely static approach, that means that the compiler should calculate, while generating the assembly code, the full mode information for each individual register for each instruction in the final code. In other words, the compiler should determine, during code generation, in which parts of the final code each register assumes each of the possible modes.

For example, let us say that we need to perform a compacting garbage collection. For each point in the code in which we want to be able to perform the operation, we would like to obtain the minimal set of registers which might contain pointers, excluding those which certainly are no longer used. Such a fine level of detail, however, can be rather difficult to obtain. The ideal situation, of course, would be having the ability to design a complete compiler system from scratch, including in its core design the distinction among the different modes, and propagating

such distinction down to the final assembly code. In practice, however, it might also make sense to try and adapt an existing system while adding the new functionalities. There can be several reasons for such an approach, first of all the need to preserve a possibly large economic investment already made while developing a certain compiler system, but also, for example, the ability to support special front ends or machine architectures, a particularly high quality of the generated code, or compatibility needs with different components. Adapting an existing compiler, on the other hand, might require some substantial work since, in certain cases, a distinction must be introduced among certain internal data types where there was none before.

Even in the ideal case in which a complete compiler can be newly designed, however, there are certain challenges that need to be addressed. A distinction among different modes in the intermediate representation can be used at the level of the internal representation, but carrying this distinction down to the level of the individual assembly instructions can require more work. In order to perform many optimisations, the compiler will rely on liveness information calculated at the level of the intermediate representation. For instance, manipulations like dead code elimination, loop invariant movements, and so on can be performed directly on the intermediate representation. Once that intermediate form is expanded in the final code, however, the liveness information previously calculated might be not detailed enough to allow for a reconstruction of the mode of all registers at each individual machine instruction.

The final stages of code generation will expand the internal constructs of the intermediate representation into possibly rather complex sequences of assembly instructions. The liveness of each register in the resulting code, will depend not just on the expansion of a single construct, but rather on the entire sequence of expansions. Some of the resulting expansions, furthermore, might be not just macro expansions, but the result of compile-time procedures that generate different code depending on the circumstances. Certain physical registers might be used or not, in the expansion for a given construct, depending on the specific code generated by the expansion rule in that particular case.

In other terms, a further liveness analysis of some sort might be required, using the real registers and the machine instructions rather than the intermediate representation. Performing such an analysis at the same time as the final code production would be far from easy, and would considerably complicate the structure of the back end. The Java compiler by Stichnoth et al., works around this issue by using an intermediate representation that has a one-to-one relationship with the individual assembly instructions [SLC99]. That means that the intermediate representation is actually, during the last stages, just an abstract representations of the final code, instruction by instruction. The advantage of such a solution is that the compiler can perform a liveness analysis on the low-level intermediate form and then reuse the same information for the machine code. On the other hand, tying the intermediate representation so strictly to the concrete machine code (x86 in that case) makes it considerably difficult to adapt the compiler to different architectures, since the separation between the intermediate form (which should be architecture-neutral, in principle) and the actual back end is lost to a great degree. Similar problems, while reconstructing the liveness of machine registers, are also present when an existing compiler is adapted in order to introduce distinctions among multiple modes.

4.2 Local annotations

In order to simplify the problem in the general case, in which a neutral intermediate representation is used, a different approach can be followed separating the code generation from the computation of the mode information. That can be achieved by generating, while the code is produced, additional annotations that refer to the way in which the registers are used locally in each expansion. Those annotations can then be parsed and postprocessed during a separate stage, simplifying the overall structure of the system.

The main point is that, if mode information is available at the level of the intermediate representation, it is normally also possible to determine, in the expansion of every construct, whether each register is locally used or defined in a certain mode within that expansion, even if there is no indication of the interactions of those occurrences with preceding or following expansions. For example, let us consider a fictional intermediate representation and an expansion rule used to load a scalar value using a sum and an array access, defined by base address, index and offset.

```
scal:A <- array:(ptr:B[scal:C+const:D])+scal:E      -- expands as:
   move B[C+D],A  %  A written as scalar, B used as ptr, C used as scalar
   add  E,A,A     %  A written as scalar, A used as scalar, E used as scalar
```

The expansion rule uses certain machine registers, represented by parameters `A`, `B` and so on, in specific modes. It is required that the register represented by `A` contains a scalar value for the rule to be applicable, and the expansion to be generated, and similarly `B` must contain a pointer, and so on (the remaining details of the fictional notation are not relevant to the example). Even though we do not know exactly which concrete machine registers will be used in place of `A`, `B` and `C` when the expansion takes place, and we do not know what happens before and after this expansion to those registers, we can still determine some local information about the way in which the registers are used. When the intermediate expression

```
scal:R3 <- array:(ptr:R5[scal:R11+40])+scal:R7
```

is expanded, for example, we can determine that register `R5` is used by the first instruction of the expansion as a pointer, `R7` is used by the second instruction as a scalar, and so on. All this local information, which results from the individual expansions, can then be written in the form of annotations to the assembly code, and later perused in order to reconstruct the full mode information. While this approach is not the only one possible, there are some advantages in postponing the mode calculation in this way. First of all, the structure of the back end remains simpler, since there is no need to embed complex mechanisms in the expansion rules in order to determine the mode information on-the-fly. Additionally, as will be shown in the following chapter (in Section 5.3), having the local information available all at once enables us to perform easily a number of important sanity checks on the way in which the registers are used by the compiler.

4.3 More details on reconstructing mode information

Reconstructing the mode information from the local annotations is basically a matter of performing a particular form of liveness analysis. We know the mode in which the registers are used and

defined at each instruction, but that information needs to be propagated to all other instructions in order to determine when each register assumes each mode throughout the code. Before the analysis can be performed, however, there are a few other aspects that should be taken into account. The first, important aspect, is how to deal with call instructions.

From the point of view of the analysis, what we need to obtain is information about the mode of registers before and after the call. If the microprocessor is stopped just before the call, the registers will be in modes that are decided by the previous instructions. If it is stopped right after the call, however, what we will see is the effect on the registers of the execution of the whole subroutine. In consequence of that call, certain registers might be overwritten, some will contain the return value and some will have been used as arguments to the call. If execution is stopped while the called subroutine is still in execution, however, the program counter will refer to a location that belongs to a different routine, and the state of the registers will depend on the mode information calculated for that routine rather than the one we are interested in at the moment.

The modified liveness analysis (intraprocedural, as we shall see shortly) can be performed quite easily even in the presence of call instructions if the total effect of the called routine can be determined statically, and the call instruction is annotated with the effects on the registers of the call as a whole. As later shown, determining such effects statically is possible thanks to the standard calling interface defined by the Application Binary Interface. Section 5.2.8 will discuss in more detail the handling of call instructions while performing the customised liveness analysis.

A microprocessor feature that can cause additional problems, while determining register modes, is the use of delay slots, as previously discussed in Section 3.7.2. The customised liveness analysis will have to consider the way in which the instructions interact in the case in which delay slots are used, and in particular the effect that delay slots have on the life of values in the registers. As we shall see later, the basic idea is to reorder, from a logical point of view, the existing instructions so that the effect on the registers can be calculated more simply. Some more work, however, will be necessary to deal with annulled delay slots and delayed call instructions.

Chapter 5 is fully devoted to a rigorous analysis of the way in which the local annotations can be used to reconstruct the mode information, and to the many details involved in dealing with delay slots (Section 5.4). The mode information, once calculated, can eventually be converted in a set of tables or similar structures (the “PC maps”), so that it can be inspected at runtime whenever a preemptive service request is received. The operation of the runtime is discussed in Chapter 8.

There is a further aspect in the determination of modes in the registers. The mode information can be determined by looking at the map associated to a single routine only for those registers that are actually used within the routine that is being examined. For some other registers there is no way of determining their mode using locally available data, yet the mode must be somehow determined. That will be done, as described in the following section and more extensively in Section 8.4.1, using information related to the routines that precede the current one in the dynamic call chain.

4.4 Prologue and epilogue

Up to now we have discussed the mode information in the main body of compiled routines. The compiler, additionally, generates special portions of code before and after each routine, the prologue and the epilogue, devoted respectively to setting up the new execution environment when the routine is called, and restoring the previous environment before returning to the caller. The main operations performed are modifying the stack pointer and/or the frame pointer and saving the registers that are defined by the Application Binary Interface as preserved across calls, but that are to be used locally by the newly called routine.

While the microprocessor is executing the prologue and the epilogue, particular care must be taken while determining the modes of registers. As previously mentioned, not all of the information needed to reconstruct the mode of every register will be available in the maps related to the current routine. The mode of some registers, not actively used by the current functions, will be the last mode that was in use in the environment of the caller. In order to clarify the situation, it will be useful to group the user-accessible registers into the following disjoint categories:

- Call-preserved registers. These registers contain, after a call to subroutine has returned, the same values that were present before the call.
- Registers volatile across calls. They may be used by the callee and they are not guaranteed to have the same value upon return. This category usually includes the registers used as arguments and as return value.
- Global registers used to store source program global variables.
- Specialised registers, handled directly by the compiler or the system.

The various categories are decided unequivocally by the Application Binary Interface for every microprocessor, in order to guarantee code interoperability. The mode of the registers in the last two categories does not depend on the executed code, since they are used in a fixed way throughout the code. Using global registers to store program global variables, in particular, is rather rare, given the necessity to maintain some sort of bookkeeping of the used registers when using separate compilation, but it may happen. The first two categories, however, describe registers whose mode can indeed change during execution, and in particular when a call to a subroutine is made.

The prologue and epilogue control creation and disposal of the environment needed for the proper execution of the routine body. It is therefore their job, among others, to save and restore registers in memory in order to adapt the specification imposed by the ABI with the use of registers made by the body. What this means in practice is that the prologue, apart from allocating the stack space needed for local use, will also save the content of some registers, typically on the stack, in order to free them for use by the callee.

To be precise, the registers saved will be those that should be preserved across calls according to the ABI, but that are reused locally by the routine body. Symmetrically, the epilogue will restore the previously saved values before returning. The caller, on its side, will save the data contained

in those registers that, according to the ABI, should be volatile, but whose contents are reused by the caller after the call. After the call the saved values can be restored into the same registers.

The mode of the registers that are volatile across calls can always be determined using locally available information, throughout the execution of each routine. The mode of those registers that are call-preserved, but which are saved by the prologue, can be determined locally within the routine body but not during the execution of prologue and epilogue. Those registers refer, at the beginning of the prologue, to the calling context and are subsequently, using one or more save operations, made available to the callee. After the save operation the registers are "owned" by the callee, but before that they are "owned" by the caller. That means that it is necessary to keep track of the exact instructions in the prologue/epilogue in which each of the registers is saved or restored. The situation is depicted in Figure 4.4.1.

In order to determine which registers are used locally and which are not, it will be necessary to use some sort of table or descriptor that describes the instructions at which the registers are saved/restored. That could be done, for instance, using bitmasks (one bit for every register which is "local" at that instruction). Alternatively, if the registers are always saved following a predetermined sequence, each instruction could be associated with the highest numbered register that was saved at that point. Other arrangements are also possible.

A simplified, slightly more conservative approach could be used considering that the saved registers are used by the routine body but not by the prologue itself. It is therefore safe to treat each of those registers as having the mode of the calling context during the whole prologue, even though they are actually unused during the last part of the prologue code. The same approach cannot always be used for the epilogue, since the values that the registers contain at the beginning

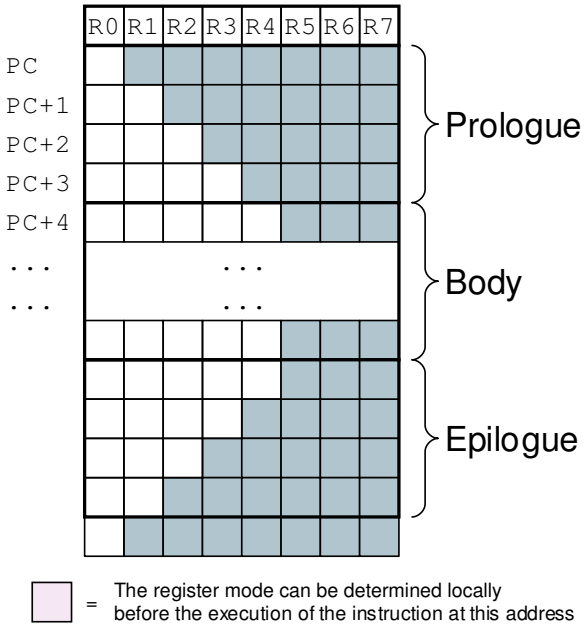


Figure 4.4.1: Local registers in prologue, body and epilogue

of the epilogue depend on what is left there by the routine body. Those values might not be consistent with the modes that the registers will later assume after their prior value is restored during the execution of the epilogue.

If the only distinction of our interest is pointer/scalar, however, that simplified approach can be safely used both in prologue and epilogue only if it is acceptable, for the service routine involved, to misidentify random values as pointers. In the initial part of the epilogue some registers will be unused and contain random data, but will be identified as pointers. Being unused, there is no problem if their content is altered. In the case of a garbage collection, for instance, the only possible effect would be potentially preventing unused blocks of memory from being reclaimed, in case the random data happens to coincide with the value of a legal pointer to a heap block.

Another complication may derive from the use of register windows, as described in Section 3.7.1. When the window shifting operation is performed, some physical registers will "change position", and the same names will refer to entirely different registers before and after the shift. Some user-accessible registers will suddenly have the mode that other registers had right before, while others will become unused, and others will remain unchanged. In the case of register windows, it is crucial to keep track of the transformation caused by the window shift, so that the correct mode can be calculated across the renaming.

A great programmer is a seeker of truth and beauty.
— Dave Winer

Chapter 5

Multi-mode Liveness Analysis and Consistency Checks

5.1 Introduction

In the context of this research, it is necessary to determine where the registers are used at each machine instruction, in optimized compiled code, as pointers or scalar values, or possibly in other ways. The different possibilities are called “modes” (Section 3.4). Each register can, at any time, be used in just a single mode, or it can be unused.

In certain cases, extracting the liveness information, at the level of the individual machine instruction, from the compiler’s internals might not be a practical solution. For instance, the generation of the final code might be done, in an existing compiler, by expanding each construct in the intermediate representation into a sequence of individual machine instructions. While the liveness information is in general available at the level of the intermediate representation, there might be no similar liveness information available for the single machine instructions. Tracing the life of the individual registers for all the instructions in the expansions during the code generation would be particularly complex, since the life of the registers could depend on the specific expansions used throughout the program, some of which might also be the result of the execution of complex expansion subroutines, called at compile-time.

A much simpler approach can be adopted if it is possible to determine, while the assembly code is being generated, which registers will be used, or overwritten, in some particular modes at each instruction. While the assembly code is generated, therefore, a series of annotations can be generated, for instance “Register R4 is used by this instruction as a pointer”, or “Register R7 is overwritten by this instruction with a scalar value”. Once the full code has been generated, it is then possible to run a separate liveness analysis involving registers and machine instructions, so that the desired liveness information can be reconstructed. Thanks to the standard calling interface provided by the Application Binary Interface, it is possible to analyse the life of registers for each

routine in isolation.

A separate liveness analysis is therefore performed using, as input, those annotations. The analysis, which is intraprocedural, is "multi-mode" in the sense that each register may assume various modes throughout the code, always one at a time. The algorithm, which will be discussed shortly, is a fairly simple adaptation of a conventional liveness algorithm. The additional twist, however, is that the compiler is not trusted to generate code that defines and uses registers in a consistent manner. That is, an erroneous code generation might cause a register to be initialized as a pointer and later used as a scalar, or initialized as a short integer and later used as a long integer. Verifying the consistency of the generated code is particularly important if, as in our case, the back end was heavily customised and new errors might have been inadvertently introduced.

In order to specify exactly the code consistency requirements, a number of desirable properties are then defined. For instance, the code should use a register in the same mode in which it was last defined. Each register should only be used after being initialized for the first time. Only if all of the properties are respected, the code is deemed valid.

The liveness analysis is actually performed regardless of the properties being satisfied. However, once the analysis is complete, the liveness information that was generated can be used to verify whether all the desired properties are indeed satisfied or not. In the following sections, it will be proved that the initial definitions of the various properties are actually equivalent to much simpler tests, which involve the (possibly meaningless) liveness information that was calculated.

As a final step, the effects of delayed control transfer are analysed. The non-trivial effects of delay slots on the reconstruction of the mode information will be discussed, and it will be shown how the liveness analysis can be performed in this case as well. The way in which that is accomplished is by building an alternative representation of the original code which makes no use of delay slots, but still sets the registers in the same modes as the original program. The previous properties can still be verified, even if delay slots are used, and the discussion shows how the previous tests can be adapted. The complexity of the whole analysis, and of the verifications of the various properties, is similar to the complexity of a normal liveness analysis.

The analysis here was specifically developed as a formal justification to the algorithms which have been used in the prototype discussed in Chapter 9. A bibliographic review of similar material did not highlight a previous discussion of the same analysis, which is therefore believed to be entirely original and presented here for the first time. Some related material, which might be of interest, is listed below.

The annotations used in this liveness analysis are reminiscent of the type annotations used by the Typed Assembly Language (TAL) [MCG+99]. The type system used by TAL is quite complex and sophisticated, and includes parametric polymorphism, existential types, recursive types, and other features. As a side effect, the type annotations can be potentially rather large in proportion to the code size, although they can be compressed to some extent [GM00]. The structures used by TAL are targeted to perform complex type verifications, unnecessary in this context.

The customised liveness analysis described later can be easily implemented using the much simpler annotations described in this work, requiring the same polynomial time as a classic liveness analysis. Although simpler, the structures used in this work are well suited to support preemptive program services. The size of the annotations is not a problem in our case.

Reconstructing the different modes that registers can assume has similarities with calculating Reaching Definitions [CK98, NNH99]. The main difference is that the Reaching Definitions analysis calculates which definitions "reach" a certain point in the code, using a forward analysis. In our case, we are also interested in determining which registers are unused at a certain point, as well as obtaining the mode of their content. The analysis performed here is a variant of the classic liveness analysis, and it uses a backward analysis.

According to Prof. Reinhard Wilhelm (Universität des Saarlandes), results similar to the ones presented here could probably be obtained, in principle, also using a form of assume-guarantee analysis [She03], although the computational complexity of such an approach might be greater. Zhichen Xu used annotations on machine code in order to explore the safety checking of machine code [XMR00]. The annotations, which refer to security conditions expressed using typestates, are then processed to verify safety properties. That approach, according to its authoR, is however not suitable to express liveness properties. The algorithms used in that work are not related to the liveness analysis presented here, but the adjustments applied in order to deal with the delay slots, required by the use of the SPARC microprocessor, are partly related to some of the mechanisms used in Section 5.4.

5.2 Multi-mode liveness analysis

5.2.1 The context

In order to give a precise description of the analysis performed on the code, a formal representation of the program code will be introduced. The same representation will be particularly useful later, in the section devoted to the consistency checks. As this discussion will focus on the intra-procedural analysis, the formal representation will model the body of a compiled routine. The term "routine" will be used, generically, to refer to any kind of procedure, method or function.

It will be assumed that a finite set of registers is available to store values, and that each register can only be used in one of a finite set of "modes" at any time. The liveness algorithm, and the subsequent checks, will determine the modes of the various registers throughout the code. The various modes, for instance pointer, short integer, long integer and so on, only loosely reflect the high-level types used by the source code, and actually only represent the way in which registers are used by the compiled assembly code. Such information can be useful to perform exact garbage collections, or to perform on-the-fly endianness conversions (for instance using double-endian microprocessors). The body of a compiled routine will be modeled as a sequence of simple instructions that use certain registers in some modes, set some registers in some modes and possibly transfer control to one of a statically known set of instructions contained in the same compiled routine.[1]

A call to a subroutine will be modeled using a single instruction that summarises the complete effect on the registers that the actual call has as a whole. A call instruction will therefore use

[1] In this chapter, the term "instructions" is used to refer to instruction occurrences; a program incrementing a register twice, using two identical increment instructions, is still considered to be made up of two distinct instructions rather than one, repeated twice.

registers (for the arguments), set registers (return values, and overwritten volatile registers) and transfer control to the following instruction (if the called subroutine can actually return control to the caller). More details on this point will be given in Section 5.2.8. It will also be assumed that it is possible to determine statically the sets of registers that can be affected (read or overwritten) by each instruction, and the modes in which those registers are used.

5.2.2 Formal definitions

A simple way to model the described structure is to use a finite set, each element of which represents a concrete instruction in the compiled subroutine, with a "successor" function to model the sequence of instructions in the code.

Definition 5.2.1. The set of instructions I is a non-empty, finite set of elements. Two elements $enter \notin I$ and $exit \notin I$ are used to represent, respectively, prologue and the epilogue of the routine, and we also define the sets $I_p = I \cup \{enter\}$, $I_e = I \cup \{exit\}$ and $I_{pe} = I_p \cup I_e$. A bijective function $succ : I_p \to I_e$ is used to represent the static sequence of instructions in the code. If a given assembler instruction is represented by i, for example, the instruction that follows in the compiled code will be represented by $succ(i)$. We require that, if $|I_p| = n$, then $succ^n(enter)$ exists and is equal to $exit$, so that all the elements of I_{pe} are effectively linked by $succ$ in a chain. The first instruction of the routine body will be represented by $i_{first} = succ(enter)$. Additionally, an element $null \notin I_{pe}$ will be used in special cases.

The definitions above model the basic static structure of the compiled routine. It is however necessary to model the dynamic behaviour as well. First of all, it is important to formalise the concept of "mode". In the previous discussions, the term mode has been used to describe informally the way in which a register is used at a certain point in the code. For instance, a certain register could be used sometimes as a pointer to data, or as a scalar value, or a pointer to code and so on. While a more precise definition of the use of registers will be given later, the different modes can be easily modeled using a finite, non-empty set. A special element, *unused*, is used to indicate that a register is not in use. The set of registers on which the algorithm works is also defined.

Definition 5.2.2. The set of modes M is a non-empty, finite set of elements. The set M_u is defined as $M_u = M \cup \{unused\}$, where $unused \notin M$. The set of registers R is a non-empty, finite set of elements: $R = \{r_0, r_1, \ldots, r_{n_R}\}$.

It is now time to describe the dynamic behaviour of the instructions, specifying the way in which each instruction affects the mode of the registers and the control flow. Each instruction, as previously mentioned, will require certain input values, with certain modes, in some registers. As a result of the execution, some registers will possibly be altered, both in value and in mode. The information available for every instruction is encoded by the following functions:

Definition 5.2.3. The function *use* describes, for each instruction, the registers required and their modes as follows: $use : I_e \times M \to 2^R$. If a machine instruction represented by $i_0 \in I$ uses a subset

of registers, represented by $R' \subseteq R$, with mode represented by $m \in M$, then $use(i_0, m) = R'$. If $i_0 = exit$, then *use* represents the registers and their modes at the end of the routine body. Similarly, the function $def : I_p \times M \rightarrow 2^R$ represents the registers that are set in a certain mode as a result of the execution of each instruction. When applied to *enter*, the function def represents the registers and their modes as supplied to the routine body by the prologue.

While this definition is mathematically consistent, our intuitive concept of mode requires that each register can be used in a single mode at any point in the code (more on that later) and such condition must be trivially verified by the definitions of def and *use* as well. Since those two functions are obtained from information supplied by the compiler, the condition should be automatically verified, but it might be useful to perform a sanity check on the input data. We require the following condition to be verified:

Condition 5.2.4. A consistent definition of def and *use* must respect the following properties:

$$\forall m_1, m_2 \in M, m_1 \neq m_2 \begin{cases} \forall i \in I_e : use(i, m_1) \cap use(i, m_2) = \varnothing \\ \forall i \in I_p : def(i, m_1) \cap def(i, m_2) = \varnothing \end{cases}$$

We will assume the definitions of def and *use* to be consistent in the following discussion, except where explicitly stated. The way in which def and *use* reflect the concrete assembler instructions is fairly intuitive for simple instructions (including branches, which on most machines do not alter registers). The situation for call instructions, as previously mentioned, is slightly more complex. While a call instruction causes a subroutine to be called, causing a certain cumulative effect on the registers, the control flow will eventually return (if the called subroutine returns at all) to the instruction following the call instruction. From the point of view of the mode analysis, the effect of the call, as a whole, can be considered similar to that of the execution of a particular kind of simple instruction that affects several registers at once. The execution flow, in general, appears locally simply to step through the call instruction as it would happen for a simple instruction. It is therefore fairly simple to handle call instructions if the subroutine's def and *use* are defined so that the cumulative effect of the called subroutine is taken into account. It is important to notice that, even if the exact routine which is to be called is not known, the argument list is statically known,[2] and the general calling convention, including the list of volatile registers, is dictated by the Application Binary Interface for the particular machine architecture. The details of those cumulative definitions will be explained in detail later on, in Subsection 5.2.8.

5.2.3 Dynamic control flow

Another important effect of instructions is the impact that each instruction has on the control flow. Depending on the particular instruction, execution can continue at the instruction immediately following, but also at a different instruction, if a branch is involved. In certain cases, an instruction might never return (for instance, calls to certain system routines or special machine instructions). The dynamic effect on the control flow is modeled by the functions $follow_j$:

[2] If variadic argument lists are used, as in the C "printf", the list of incoming arguments is not statically known to the callee. However, the list of outgoing arguments at a particular call site is still statically known to the caller.

Definition 5.2.5. If, for a given routine, instructions can transfer control to a maximum of n_f distinct instructions where $n_f \leq n = |I_e|$, then the functions $follow_j : I_p \to I_e \cup \{null\}$ are defined for each $j = 0, \ldots, n_f - 1$, representing the possible instruction that can follow a given one (or *null* if the number of alternatives is less than $n_f - 1$).

For example, if i is an ordinary non-branching instructions, we will have that $follow_0(i) = succ(i)$, while $\forall j = 1, \ldots, n_f - 1 : follow_j(i) = null$. If i is a conditional branch, we will define $follow_0(i) = succ(i)$, and $follow_1(i)$ to specify the branch target. If jump tables are not used, and there are no indirect jumps, then $n_f = 2$ is usually sufficient to specify the possible destinations of all instructions (as said, we treat calls as a special case of non-branching instructions). The use of I_p and $I_e \cup \{null\}$, respectively, as domain and codomain of the functions $follow_j$, represents the fact that the prologue cannot be the destination of any control transfer instruction and that, once the epilogue has been reached, the control flow cannot return to the routine body. We can now begin our discussion with a few considerations and extensions on the functions $follow_j$.

Definition 5.2.6. $follow : 2^{I_{pe}} \to 2^{I_{pe}}$ is defined as follows:

$$\forall A \subseteq I_{pe} : follow(A) = \bigcup_{\forall i \in A \setminus \{exit\}} \left\{ follow_j(i) \mid j = 0, \ldots, n_f - 1 \right\} \setminus \{null\}$$

The function $follow$ represents the set of instructions that are dynamically reachable in one step from another set of instructions.

Since the codomain of $follow$ is the same as its domain, the notations $follow^k$, $follow^+$ and $follow^\star$ can be used with the usual meaning. For instance, if $k \in \mathbb{N}^+$ and $A \subseteq I_{pe}$, the function $follow^k(A)$ is defined as $follow^{k-1}(follow(A))$, where $follow^0(A) = A$. The function $follow^+$is defined as $\forall A \subseteq I_{pe} : follow^+(A) = \bigcup_{\forall k \in \mathbb{N}^+} follow^k(A)$ and $follow^\star$ as $\forall A \subseteq I_{pe} : follow^\star(A) = \bigcup_{\forall k \in \mathbb{N}} follow^k(A)$. An instruction $i_1 \in I_{pe}$ is said to be *reachable* from instruction $i_0 \in I_{pe}$ iff $i_1 \in follow^\star(\{i_0\})$. An instruction is simply said to be *reachable* if it is reachable from *enter*.

Intuitively, since we are dealing with finite sets, the function $follow^\star$ can be calculated in a finite time following all possible paths using any graph traversal algorithm. For completeness, an iterative algorithm that can be used to calculate $follow^\star$is briefly discussed.

Lemma 5.2.7. *follow is monotonic.*

Proof. Let us consider $A \subseteq B \subseteq I_{pe}$. The following relation holds:

$$\bigcup_{\forall i \in A \setminus \{exit\}} \left\{ follow_j(i) \mid j = 0, \ldots, n_f - 1 \right\} \subseteq \bigcup_{\forall i \in B \setminus \{exit\}} \left\{ follow_j(i) \mid j = 0, \ldots, n_f - 1 \right\}$$

which implies $follow(A) \subseteq follow(B)$. □

Definition 5.2.8. Operator $S : \left(2^{I_{pe}} \to 2^{I_{pe}}\right) \to \left(2^{I_{pe}} \to 2^{I_{pe}}\right)$ is defined as:

$$\forall f : 2^{I_{pe}} \to 2^{I_{pe}}, \forall A \subseteq I_{pe} : S(f)(A) = A \cup follow(f(A))$$

A partial order is now introduced among the functions $2^{I_{pe}} \to 2^{I_{pe}}$.

Definition 5.2.9. The partial order $\trianglelefteq$ on $2^{I_{pe}} \to 2^{I_{pe}}$ is defined as:

$$\forall f,g : 2^{I_{pe}} \to 2^{I_{pe}} : f \trianglelefteq g \Leftrightarrow \forall A \subseteq I_{pe} : f(A) \subseteq g(A)$$

The partial order $\trianglelefteq$, together with the set $2^{I_{pe}} \to 2^{I_{pe}}$ forms a lattice, which is finite and therefore also complete.

Proposition 5.2.10. *Operator S is monotonic over the lattice* $\left(\trianglelefteq, 2^{I_{pe}} \to 2^{I_{pe}}\right)$.

Proof. Let us consider two functions $f,g : 2^{I_{pe}} \to 2^{I_{pe}}$ such that $f \trianglelefteq g$. We have $\forall A \subseteq I_{pe} : f(A) \subseteq g(A)$, and consequently $\forall A \subseteq I_{pe} : A \cup follow(f(A)) \subseteq A \cup follow(g(A))$ since $follow$ is monotonic. Hence $S(f) \trianglelefteq S(g)$. □

Since S is a monotonic operator defined over a complete lattice, it has a unique least fixed point according to Tarski's fixed point theorem.

Proposition 5.2.11. *Function* $follow^\star : 2^{I_{pe}} \to 2^{I_{pe}}$ *is the least fixed point of operator S over* $\left(\trianglelefteq, 2^{I_{pe}} \to 2^{I_{pe}}\right)$.

Proof. $follow^\star$ is a fixed point. In fact,

$$\forall A \subseteq I_{pe} : A \cup follow(follow^\star(A)) = follow^0(A) \cup follow^+(A) = follow^\star(A)$$

Furthermore, for every fixed point g, we can show that $follow^\star \trianglelefteq g$. From the definition of S, for every function $g : 2^{I_{pe}} \to 2^{I_{pe}}$, it is trivially

$$\forall k \in \mathbb{N}, \forall A \subseteq I_{pe} : S^k(g)(A) = follow^k(g(A)) \cup \bigcup_{\forall j \in \{0,\ldots,k-1\}} follow^j(A)$$

and, if g is a fixed point, $\forall k \in \mathbb{N} : S^k(g) = g$. Therefore, $\forall k \in \mathbb{N}, \forall A \subseteq I_{pe} : follow^k(A) \subseteq g(A)$, and therefore, if $i \in follow^\star(A)$, it is also $i \in g(A)$, and therefore $follow^\star \trianglelefteq g$. Summarising, $follow^\star$ is a fixed point and, for every fixed point g, we have $follow^\star \trianglelefteq g$, hence $follow^\star$ is the least fixed point. □

In order to calculate $follow^\star$, it is consequently sufficient to apply iteratively operator S over the function that has the empty set as every value, until a fixed point is reached (which happens in a finite number of steps, since the number of possible functions is finite).

5.2.4 Expected mode

Informally, a register $r \in R$ has "expected mode" $m \in M$ for instruction $i \in I_e$ if register r is used with that mode by i, or alternatively by any instruction that can follow i along at least one possible dynamic execution path, if the register is not redefined between i (inclusive) and that use of r. If the condition cannot be verified for any $m \in M$, r is not expected to be in any particular mode by

i. If every register is in its expected mode at every instruction, then all the instructions will find the registers in the required modes for their execution.

The definition above allows for a register to be in multiple modes at once and does not require those registers to be set in the correct modes by other instructions. We will later require each use to be matched by a register definition, and each register to be in only one mode for each instruction. Additionally, we will point out that the mode requirements can be safely ignored for those instructions that can never be reached by the control flow. It should also be noted out that the definition above for "expected mode" takes into account every possible execution path in order to allow for a static analysis, and it only represents an approximation over the real, dynamic behaviour. The intuitive and rather informal definition above can be better phrased, in formal terms, using the functions defined so far.

Definition 5.2.12. Function $expected : I_e \times M \to 2^R$ is defined as follows:

$$\forall i \in I_e, \forall r \in R, \forall m \in M :$$
$$r \in expected\,(i,m) \Leftrightarrow \exists k \in \mathbb{N} : \exists i_0, \ldots, i_k \in I_{pe} : \left(i_0 = i \wedge r \in use(i_k, m) \wedge \right.$$
$$\left. \forall j \in \{0, \ldots, k-1\} : \left(i_{j+1} \in follow\left(\{i_j\}\right) \wedge r \notin \bigcup_{\forall m' \in M} def\,(i_j, m') \right) \right)$$

A register r belongs to $expected\,(i,m)$ if i or a following instruction uses register r in mode m, and there is no definition in i or afterwards encountered before that use. The definition summarises the mode in which all registers are expected to be before the execution of every instruction. Other conditions, however, must be verified before the definition above can be used on a practical level. Since many of those conditions will be used several times, some will be given a conventional name.

Definition 5.2.13. A routine body is "sufficient" if each register expected in a certain mode, for any reachable instruction, is set in that mode by at least one preceding definition along every possible execution path. Formally:

$$\forall m \in M, \forall r \in R, \forall k \in \mathbb{N}^+ : \forall i_0, \ldots, i_k \in I_{pe} :$$
$$\left(i_0 = enter \wedge \left(\forall j \in \{0, \ldots, k-1\} : i_{j+1} \in follow\left(\{i_j\}\right)\right) \wedge\right.$$
$$\left. r \in expected\,(i_k, m) \Rightarrow \exists j \in \{0, \ldots, k-1\} : r \in def\,(i_j, m)\right)$$

If a routine body is not sufficient, some instructions might not have their registers set in the expected mode during execution, which also means that some uninitialised registers might be used by the program. While being sufficient is an important property of the function body, it is also important to verify whether the various definitions along every dynamic path are coherent with the expected mode, which is a stronger property. The following, rather intricate definition will completely describe the desired property.

Definition 5.2.14. A routine body is "valid" if, for every instruction that sets a register, and for every instruction that, following some execution path from the previous instruction, expects the

same register to be in one given mode, the mode in which the register is set is the same in which it is expected. Formally:

$$\forall i \in I_e, \forall m \in M, \forall r \in expected\,(i,m)\,, \forall i_d \in follow^{\star}\,(\{enter\}) \setminus \{exit\} :$$
$$\left(r \in \bigcup_{\forall m' \in M} def\,(i_d, m') \wedge \exists k \in \mathbb{N}^{+} : \exists i_0, \ldots, i_k \in I_{pe} : \left(i_0 = i_d \wedge i = i_k \wedge \right.\right.$$
$$\left.\left. \forall j \in \{0, \ldots, k-1\} : i_{j+1} \in follow\left(\{i_j\}\right) \wedge \forall j \in \{1, \ldots, k-1\} : r \notin \bigcup_{\forall m' \in M} def\,(i_j, m') \right) \right) \Rightarrow$$
$$r \in def\,(i_d, m)$$

The above definition, apparently rather complicated, can be easily decoded as follows: if a register is expected in a given mode, a valid definition must ensure that for every i_d that precedes i, and which defines that register in any mode while not being followed by other definitions, the mode in which i_d sets register r must be exactly m. An alternative, simpler condition equivalent to validity will be discussed later. The restriction on the set of possible values for i_d ensures that the effect of those instructions which are in unreachable portions of the routine body are actually ignored. While validity ensures that all register mode definitions are suitable for their successive use, there is no guarantee that a definition exists for every use.

As an interesting observation, it should be noted that, while a valid routine body is guaranteed to have every register used by *use* matched correctly by its definitions, it may actually happen that some register mode definitions are not matched by any *use*. That can happen, for instance, if a function is passed more parameters than necessary for its execution. The extra parameters will appear in $def\,(enter, m)$ but, not being used, will not appear in $expected\,(i_{first}, m)$. Such an occurrence is not necessarily an indication of a problem, since there might be legitimate reasons for the presence of the additional parameters even in optimised code (for instance, backward compatibility needs).

Another case in which some registers may appear to be defined but never used is the use of function calls as single instructions. As described in detail in Section 5.2.8, call instructions can be considered as having an effect similar to that of single instructions. During the call, however, some registers can have their value overwritten even if they do not return any useful information to the caller. This overwriting can be represented as a *def* of the corresponding registers in a special additional mode, without any matching *use*.

Definition 5.2.15. A routine body is "correct" if it is both sufficient and valid.

If a routine body is correct, then every register mode definition sets the register in the mode which is expected by the following instructions, and at least one valid mode definition is present for every use along every execution path. Another important requirement of programs asserts that each register must be, at any point in the program, in only one of the modes $m \in M$.

Definition 5.2.16. A routine body will be called "consistent" if each register is only ever used in a single mode at each point in the code:

$$\forall i \in follow^{+}\,(\{enter\})\,, \forall m_1, m_2 \in M, m_1 \neq m_2 : (expected\,(i, m_1) \cap expected\,(i, m_2)) = \varnothing$$

Proposition 5.2.17. *If a routine body is correct, it is also consistent.*

Proof. It that were not the case, there would be an r such that $r \in expected\,(i,m_1) \cap expected\,(i,m_2)$, $m_1 \neq m_2$. However, by Definition 5.2.15, the function body is also valid, and by Definition 5.2.14 we would require to have a mode definitions for r in both modes simultaneously, that is $r \in def\,(i_d,m_1) \cap def\,(i_d,m_2)$ (and we must be able to find one such i_d since the body is also sufficient by the definition of correctness). However, there can be no register in two different modes in def for the same instruction because of Condition 5.2.4, hence the conclusion is absurd. Every function body that is correct is therefore also consistent. □

In a program that is correct, and consequently consistent, every instruction is guaranteed to find the values it requires set in the correct modes and each register is used, at any given time, in a single mode. In compilers, each register is usually assigned, at every position in the code, to a single pseudo-register, which generally ensures that a single mode is used at each point in the code (this is the case with GCC). While it can be usually assumed that consistency, and even correctness, are automatically satisfied in compiler-generated programs, it can be useful to verify explicitly the relevant properties for added safety. It should be noted that Condition 5.2.4 (consistent definition of def and *use*) can be in practice relaxed to take into account only reachable instructions since unreachable instructions do not influence consistency, according to the proof to the previous proposition.

It is quite important to point out that both properties, correctness and consistency, are not strictly necessary to guarantee that a program will work as intended. It is fairly easy to write a hand-coded piece of assembly code that, despite appearing inconsistent or incorrect, still works as intended because of the particular nature of the program logic. On the other hand, a program that is both correct and consistent is certainly "safe" from the point of view of the static mode analysis since, whatever the real dynamic execution path, all instructions will be supplied with all the necessary registers in the correct mode.

5.2.5 Mode calculation

As we have seen, if a program is correct and consistent, the register modes represented by the function *expected* can be used to describe the use of registers, and the modes in which they can be used, in the body of the routine. In particular, in a consistent routine body, according to Definitions 5.2.12 and 5.2.16, each register can be expected at each reachable instruction in one mode at most. In fact, $\forall i \in follow^+(\{enter\})$, $\forall r \in R$, if $\exists m \in M : r \in expected\,(i,m)$, by Def. 5.2.16 the same registers cannot appear in any $expected\,(i,m')$ with $m' \in M, m \neq m'$.

It is therefore possible to select the only m such that $r \in expected\,(i,m)$ or the element *unused* to create, for added convenience, a function $mode : I_e \times R \to M_u$ as follows:

Definition 5.2.18. The function $mode : I_e \times R \to M_u$ is defined on consistent routine bodies as:

$$\forall r \in R : \begin{cases} \forall i \in follow^{+}(\{enter\}) : \begin{cases} \forall m \in M : (r \in expected(i,m) \Rightarrow mode(i,r) = m) \\ \nexists m \in M : r \in expected(i,m) \Rightarrow mode(i,r) = unused \end{cases} \\ \forall i \in I_e \setminus follow^{+}(\{enter\}) : mode(i,r) = unused \end{cases}$$

where *mode* is well-defined, as previously shown. For all unreachable instructions, the real value of *expected* has no relevance. The function *mode* is therefore conventionally set to *unused*, to indicate that all unreachable instructions do not concretely require the registers to be in any particular mode.

It will now be our task to calculate the function *expected*, and consequently *mode*, algorithmically. From the definition of *expected*, in Definition 5.2.12, it is intuitively apparent that the mode calculation can be performed propagating backwards, along all the possible dynamic execution paths, the mode requirements of each instruction specified by the function *use*, until a matching mode definition, described by *def*, is encountered. In that sense the mode calculation algorithm is simply a variant of the well-known liveness analysis that is commonly performed during compilation, and can be shown to have similar complexity and termination properties.

In order to show how *expected* can be calculated iteratively, it will be useful to introduce a new operator as follows:

Definition 5.2.19. Operator $T : \left(I_e \times M \to 2^R\right) \to \left(I_e \times M \to 2^R\right)$ is defined as:

$$\forall f : I_e \times M \to 2^R, \forall i \in I_e, \forall m \in M, \forall r \in R :$$
$$r \in T(f)(i,m) \Leftrightarrow r \in use(i,m) \vee \left(r \notin \bigcup_{\forall m' \in M} def(i,m') \wedge r \in \bigcup_{\forall i' \in follow(\{i\})} f(i',m) \right)$$

Operator T encodes the relationship between the modes expected by instructions and the modes expected by their followers. A partial order among the functions $I_e \times M \to 2^R$ is now introduced:

Definition 5.2.20. The partial order $\sqsubseteq$ on $I_e \times M \to 2^R$ is defined as follows:

$$\forall f, g : I_e \times M \to 2^R : f \sqsubseteq g \Leftrightarrow (\forall i \in I_e, \forall m \in M : f(i,m) \subseteq g(i,m))$$

The partial order $\sqsubseteq$, together with the set $I_e \times M \to 2^R$, forms a lattice, which is finite and therefore also complete. The simple function z, defined as $\forall i \in I_e, \forall m \in M : z(i,m) = \varnothing$, is the lattice bottom since $z \sqsubseteq f$ for every function f in the set $I_e \times M \to 2^R$.

Proposition 5.2.21. *Operator T is monotonic in the lattice defined by Definition 5.2.20.*

Proof. Let us consider $f, g : I_e \times M \to 2^R$ such that $f \sqsubseteq g$. That means $\forall i \in I_e, \forall m \in M, \forall r \in R :$ $r \in f(i,m) \Rightarrow r \in g(i,m)$. Let us prove that $T(f) \sqsubseteq T(g)$. If $r \in T(f)(i,m)$, then

$$r \in use(i,m) \vee \left(r \notin \bigcup_{\forall m' \in M} def(i,m') \wedge r \in \bigcup_{\forall i' \in follow(\{i\})} f(i',m) \right)$$

If $r \in use(i,m)$, then $r \in T(g)(i,m)$. Otherwise, $r \in \bigcup_{\forall i' \in follow(\{i\})} f(i',m)$. But each of the sets $f(i',m)$ are included in the corresponding $g(i',m)$, therefore $r \in \bigcup_{\forall i' \in follow(\{i\})} g(i',m)$. Consequently, in this case as well, $r \in T(g)(i,m)$. Therefore $\forall i \in (I \cup \{exit\}), \forall m \in M, \forall r \in R : r \in T(f)(i,m) \Rightarrow r \in T(g)(i,m)$, which means $T(f) \sqsubseteq T(g)$. □

Since T is monotonic over a complete lattice, it has a unique least fixed point, according to Tarski's fixed point theorem. Furthermore, such a least fixed point can be obtained iterating T over the lattice bottom. Let h be the least fixed point of T over the lattice $\left(\sqsubseteq, I_e \times M \to 2^R\right)$. We intend to show that $h = expected$.

Lemma 5.2.22. *expected is a fixed point of T.*

Proof. We must show that $T(expected) = expected$. Before proceeding, it is useful to review the complete, expanded expressions for *expected* and $T(expected)$.

- $\forall i \in I_e, \forall m \in M, \forall r \in R$, if $r \in expected(i,m)$, by Def. 5.2.12:

$$\exists k \in \mathbb{N} : \exists i_0, \ldots, i_k \in I_{pe} : \left(i_0 = i \wedge r \in use(i_k, m) \wedge \right.$$
$$\left. \forall j \in \{0, \ldots, k-1\} : \left(i_{j+1} \in follow\left(\{i_j\}\right) \wedge r \notin \bigcup_{\forall m' \in M} def(i_j, m') \right) \right)$$

- $\forall i \in I_e, \forall m \in M, \forall r \in R$, if $r \in T(expected)(i,m)$, by Def. 5.2.19:

$$r \in use(i,m) \vee \left(r \notin \bigcup_{\forall m' \in M} def(i,m') \wedge r \in \bigcup_{\forall i' \in follow(\{i\})} expected(i',m) \right)$$

The latter expression can be transformed in a few steps to show that it is indeed equivalent to the former. First of all, if $r \in \bigcup_{\forall i' \in follow(\{i\})} \{r' \in R \mid p(r')\}$ it also means that $\exists i' \in follow(\{i\}) : p(r)$.

Expanding $expected(i',m)$, the expression can be rewritten as:

$$r \in use(i,m) \vee \left(r \notin \bigcup_{\forall m' \in M} def(i,m') \wedge \exists i' \in follow(\{i\}) : \right.$$
$$\exists k \in \mathbb{N} : \exists i_0, \ldots, i_k \in I_{pe} : \left(i_0 = i' \wedge r \in use(i_k, m) \wedge \right.$$
$$\left. \left. \forall j \in \{0, \ldots, k-1\} : \left(i_{j+1} \in follow\left(\{i_j\}\right) \wedge r \notin \bigcup_{\forall m' \in M} def(i_j, m') \right) \right) \right)$$

which is trivially equivalent to the first expression. We can conclude that $T(expected) = expected$, therefore *expected* is a fixed point for T. □

In order to show that $h = expected$, we only need to show that $expected \sqsubseteq h$. We will show that, for every fixed point g of T, we have $expected \sqsubseteq g$.

Lemma 5.2.23. *expected* $\sqsubseteq h$.

Proof. Let us assume that $r \in expected\,(i,m)$. According to the definition of *expected*, $\exists i_k \in follow^k(\{i\})$ which satisfies $r \in use\,(i_k,m)$ and there are no definitions between i and i_k, following the chain described by $follow$. Since $r \in use\,(i_k,m)$, we know that, for every function $g : I_e \times M \to 2^R$, we have $r \in T\,(g)\,(i_k,m)$. If g is a fixed point, that also means $r \in g\,(i_k,m)$. From the definition of *expected*, we obtain that $\forall j = 0,\ldots,k-1 : r \notin \bigcup_{\forall m' \in M} def\,(\{i_j\},m')$. From $r \in g\,(i_k,m)$ and $r \notin \bigcup_{\forall m' \in M} def\,(\{i_{k-1}\},m')$, the definition of T implies that $r \in T\,(g)\,(i_{k-1},m)$. Since g is a fixed point, $r \in g\,(i_{k-1},m)$ and, proceeding along the sequence, $\forall j = 0,\ldots,k : r \in g\,(i_l,m)$, which also means $r \in g\,(i_0,m)$ and therefore $r \in g\,(i,m)$. Hence, for every fixed point g of T, it is $expected \sqsubseteq g$, which also implies $expected \sqsubseteq h$. □

Proposition 5.2.24. *expected is the least fixed point of operator* T .

Proof. Consequence of Lemma 5.2.23and Lemma 5.2.22: the least fixed point h must be unique, and since $expected \sqsubseteq h$ and *expected* is a fixed point, *expected* must be identical to h. □

5.2.6 The mode algorithm

The previous lengthy discussion shows how *expected* can be calculated iteratively applying repeatedly operator T over the function that has the empty set as every value. The following algorithm fills in the values of the maps representing *expected* (shown as *exp* below) using def, *use*, and $follow_j$ for every $j = 0,\ldots,n_f - 1$.

Algorithm 1 – Mode calculation algorithm

procedure *computeModes*()
Uses: Map def defined on $I_p \times M$
Uses: Map *use* defined on $I_e \times M$
Uses: Maps $follow_j$ defined on I_p for every $j = 0,\ldots,n_f - 1$
Calculates: Map *exp* defined on $I_e \times M$

{Initialisation of *exp*}
1: **for all** $i \in I_e$ **do**
2: **for all** $m \in M$ **do**
3: $exp\,[i,m] \leftarrow use\,[i,m]$
4: **end for**
5: **end for**
{$\bigcup def$ is precalculated}
6: **for all** $i \in I$ **do**
7: $r\,[i] \leftarrow \varnothing$
8: **for all** $m \in M$ **do**
9: $r\,[i] \leftarrow r\,[i] \cup def\,[i,m]$
10: **end for**

11: **end for**
12: **repeat**
13: $allDone \leftarrow true$
14: **for all** $i \in I$ **do**
15: **for all** $m \in M$ **do**
16: $f \leftarrow \varnothing$
17: **for all** j such that $0 \leq j < n_f$ **do**
18: **if** $follow_j[i] \neq null$ **then**
19: $f \leftarrow f \cup exp[follow_j[i], m]$
20: **end if**
21: **end for**
22: $exp1 \leftarrow use[i,m] \cup (f \setminus r[i])$
23: **if** $exp1 \neq exp[i,m]$ **then**
24: $allDone \leftarrow false$
25: $exp[i,m] \leftarrow exp1$
26: **end if**
27: **end for**
28: **end for**
29: **until** $allDone$

A concrete implementation of the above algorithm can be easily optimised during implementation. For instance, since the propagation takes place backwards, the "for" loop in line 14 is more efficiently performed scanning I from the last instruction to the first (the element *exit* is not processed, since it has no followers). At the end of the execution, *exp* contains the representation of function *expected*, from which *mode* (if the body is consistent) can be easily determined.

5.2.7 Termination and complexity

The core of the mode calculation algorithm (Algorithm 1) is a straightforward implementation of the operator T. Because of the previous discussions, the algorithm always terminates finding a unique least fixed point, which is exactly *expected*. As far as the complexity is concerned, each iteration of the main loop of the algorithm must change *exp*, but always monotonically. The number of possible maps is $|I_e| \times |M| \times \left|2^R\right|$, therefore the number of iterations must be lower than that value. During each iteration, the number of operations is proportional to $|I| \times |M| \times n_f$, therefore the algorithm, in the worst case, has maximum time complexity $O\left(|I|^3\right)$, since we can consider the sets M and R as constant for a given architecture, and $n_f \leq |I_p|$ (and $|I_p| = |I| + 1$). Notably, if no jump tables are used, there are only two possible followers for each instruction and the maximum complexity drops to $O\left(|I|^2\right)$.

It is important to point out that, despite the maximum complexity, the maximum length of a typical routine body is limited by practical reasons. In any programming language, extremely long subroutines tend to be quite rare, because of the inherent difficulty in writing and maintaining

them. An exception would be automatically generated code. In practical terms, the time required for the algorithm execution in the test implementation, on the test programs tried, has been extremely modest. By comparison, the worst case time complexity of a general liveness analysis is $O\left(N^4\right)$, where N is the size of the program, since it is assumed that there could be up to N possible variables (one variable defined at every instruction).

As far as space complexity is concerned, the only significant space is used by the maps used as arguments, or the map used for *expected*. The use of the temporary map r requires space $O\left(|I|\right)$, but the algorithm could be easily changed to run in constant space, calculating the elements of $\bigcup def$ in the main loop as necessary.

5.2.8 The effect of calls

As previously mentioned, each call instruction can be treated as a single instruction, from the point of view of the local mode analysis. The rationale behind this possibility, and the way in which registers are used across calls, will now be discussed.

In general, when a routine is invoked, by means of a call instruction, some registers will be used according to specific conventions depending on the specific function called. Certain registers will be needed by the callee to store some of the incoming parameters, and some registers will be set as return values. Some registers will not be preserved unchanged by the callee while others could be used to store temporary values, overwriting the previous content, and those registers will have no any meaningful value for the caller upon return. If it is necessary to deal with arbitrary routines hand-coded in assembler language, determining the way in which each register is used might require a detailed inspection of the called routine and, if other routines are in turn called, a complex inter-procedural analysis.

In modern computer systems, however, every routine, regardless of whether it is automatically generated by a compiler or hand-coded, is expected to conform, for any specific microprocessor architecture, to a set of detailed specifications known as Application Binary Interface (ABI). The ABI, defining a uniform coding standard and standard calling conventions, ensures full interoperability among code generated using different tools, so that libraries, system routines and user code can freely call each other. In particular, the ABI for a specific architecture usually defines which registers are used to store incoming parameters, which the return values, which registers must be preserved unchanged across function calls and which are assumed to be volatile, and can be freely overwritten.

As a consequence, all the information about the use of registers can be determined by analysing the caller code in isolation. The number, order and mode of arguments, and the mode of the return values can be determined if the signature of the called function is available, while the distinction between volatile and preserved registers is made by the ABI, and is valid for every routine. A small diagram, representing an example situation, may help to clarify the use of the various registers.

In the chosen example, shown in Figure 5.4.1, all routines can overwrite, according to the ABI, registers r0...r4, while they must preserve the remaining registers. From the signature, we know that registers r0...r3 are user as arguments, and r0 contains, after the call, the return value.

Using such information, it is possible to consider the effect of the call, from the point of view of the mode analysis, as equivalent to the effect of a single pseudo-instruction, which does not change the control flow and which uses registers r0 and r3 as integers, uses registers r1 and r2 as pointers, returns an integer value in r0 and may overwrite with some unknown values the content of r1...r4. Crucially, all the necessary information can be obtained without inspecting the actual function that is being called. The equivalent pseudo-instruction can then be used, without further complications, in the mode analysis as any other simple instruction.

Quite interesting is the treatment reserved to those registers which can be overwritten by the callee, but which do not contain any useful information when the called function returns. It is necessary to mark them as possibly overwritten, while asserting that they are not in any of the modes normally used for the registers. The goal can be easily accomplished by adding to the pre-existing set of modes M_u one new mode *volatile*, with $volatile \notin M_u$. The fact that volatile registers can be overwritten is then simply described by adding those registers to a new $def\,(i, volatile)$. Such a definition cannot match any *use*, yet it obscures previous register mode definitions from other instructions.

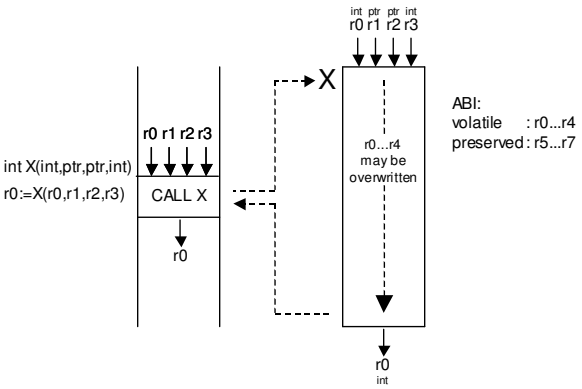


Figure 5.2.1: Cumulative effect of calls

If i is the call instruction in the example above, and M was originally $\{int, pointer\}$, the *def* and *use* functions of the equivalent pseudo-instruction i' will become:

$$\begin{array}{lcl;{4pt/4pt}lcl} use\,(i', int) & = & \{r0, r3\} & def\,(i', int) & = & \{r0\} \\ use\,(i', pointer) & = & \{r1, r2\} & def\,(i', pointer) & = & \varnothing \\ use\,(i', volatile) & = & \varnothing & def\,(i', volatile) & = & \{r1, r2, r3, r4\} \end{array}$$

Similar considerations also apply if the call instruction makes use of delay slots, as will be discussed later in Subsection 5.4.7.5.

5.3 Sanity checks

5.3.1 Additional checks

It is generally quite important to verify whether a function body, processed with the mode calculation algorithm, is consistent or correct. While the test for consistency is quite simple, according to Definition 5.2.16, verifying whether a function body is correct, referring to Definition 5.2.15, is not particularly easy. It may be quite useful, therefore, to devise an alternative test that can offer us similar information.

Condition 5.3.1.

$$\forall m_1, m_2 \in M, m_1 \neq m_2 : \forall i \in I_p, \forall i_f \in follow(\{i\}) : def(i, m_1) \cap expected(i_f, m_2) = \varnothing$$

Using a few lemmas, it will be possible to show that the simple Condition 5.3.1 is actually equivalent to validity, by rewriting and successively simplifying Definition 5.2.14. The transformation is somewhat complex, however, and requires some preliminary work. To simplify the notation, we will define I_r as $follow^{\star}(\{enter\}) \setminus \{exit\}$, and slightly modify the original definition of validity to obtain the following equivalent expression:

$$\forall i \in I_e, \forall m \in M, \forall r \in R : r \in expected(i, m) \Rightarrow \Bigg(\forall i_d \in I_r : \Bigg(r \in \bigcup_{\forall m' \in M} def(i_d, m') \wedge \exists k \in \mathbb{N}^+ : \exists i_0, \ldots, i_k \in I_{pe} : \Bigg(i_0 = i_d \wedge i = i_k \wedge \forall j \in \{0, \ldots, k-1\} : i_{j+1} \in follow\left(\{i_j\}\right) \wedge \forall j \in \{1, \ldots, k-1\} : r \notin \bigcup_{\forall m' \in M} def(i_j, m') \Bigg) \Bigg) \Rightarrow r \in def(i_d, m) \Bigg)$$

In order to simplify the next stages of the transformation, the following lemma will be useful.

Lemma 5.3.2. $\forall i_d \in I_r, \forall r \in R, \forall m \in M$:

$$\Bigg(\exists i \in I_e : \Bigg(r \in expected(i, m) \wedge \exists k \in \mathbb{N}^+ : \exists i_0, \ldots, i_k \in I_{pe} : \Bigg(i_0 = i_d \wedge i = i_k \wedge \forall j \in \{0, \ldots, k-1\} : i_{j+1} \in follow\left(\{i_j\}\right) \wedge \forall j \in \{1, \ldots, k-1\} : r \notin \bigcup_{\forall m' \in M} def(i_j, m') \Bigg) \Bigg) \Leftrightarrow r \in \bigcup_{\forall i_f \in follow(\{i_d\})} expected(i_f, m) \Bigg)$$

Proof. Expanding the definition of $expected(i, m)$, the first part of the double implication above can be rewritten as:

$$\exists i \in I_e : \Bigg(\Bigg(\exists k \in \mathbb{N} : \exists i_0, \ldots, i_k \in I_{pe} : \Bigg(i_0 = i \wedge r \in use(i_u, m) \wedge \forall j \in \{0, \ldots, k-1\} : i_{j+1} \in follow\left(\{i_j\}\right) \wedge \forall j \in \{0, \ldots, k-1\} : r \notin \bigcup_{\forall m' \in M} def(i_j, m') \Bigg) \Bigg) \wedge \Bigg(\Bigg(\exists k \in \mathbb{N}^+ : \exists i_0, \ldots, i_k \in I_{pe} : \Bigg(i_0 = i_d \wedge i = i_k \wedge j \in \{0, \ldots, k-1\} : i_{j+1} \in follow\left(\{i_j\}\right) \wedge \forall j \in \{1, \ldots, k-1\} : r \notin \bigcup_{\forall m' \in M} def(i_j, m') \Bigg) \Bigg) \Bigg) \Bigg)$$

The two large subexpressions are similar, and can be combined. Since the symbol k is already used in the first subexpression, the quantification $\exists k \in \mathbb{N}^+$will be renamed as $\exists q \in \mathbb{N}^+$. The simbols $i_0,\ldots,i_k$ will be renamed as $i'_0,\ldots,i'_q$ in the second part.

$$\exists i \in I_e : \left(\left(\exists k \in \mathbb{N} : \exists i_0,\ldots,i_k \in I_{pe} : \left(i_0 = i \wedge r \in use(i_u,m) \wedge \right.\right.\right.$$
$$\left.\left.\left.\forall j \in \{0,\ldots,k-1\} : i_{j+1} \in follow\left(\{i_j\}\right) \wedge \forall j \in \{0,\ldots,k-1\} : r \notin \bigcup_{\forall m' \in M} def(i_j,m')\right)\right) \wedge\right.$$
$$\left(\left(\exists q \in \mathbb{N}^+ : \exists i'_0,\ldots,i'_q \in I_{pe} : \left(i'_0 = i_d \wedge i = i'_q \wedge \right.\right.\right.$$
$$\left.\left.\left.\left.j \in \{0,\ldots,q-1\} : i'_{j+1} \in follow\left(\left\{i'_j\right\}\right) \wedge \forall j \in \{1,\ldots,q-1\} : r \notin \bigcup_{\forall m' \in M} def(i_j,m')\right)\right)\right)\right)$$

Now we define $k' = k+q-1$ and a single sequence $z_0,\ldots,z_{k'}$ as follows:

$$z_0 = i'_1,\ldots,z_{q-1} = i'_q = i = i_0,\ldots,z_{q+k-1} = z_{k'} = i_k$$

which also means that $z_0 = i'_1 \in follow(\{i_d\})$. We obtain the following combined expression:

$$\exists i \in I_e : \left(\left(\exists k' \in \mathbb{N} : \exists z_0,\ldots,z_{k'} \in I_{pe} : z_0 \in follow(\{i_d\}) \wedge r \in use(z_{k'},m) \wedge \right.\right.$$
$$\left.\left.\forall j \in \{0,\ldots,k'-1\} : z_{j+1} \in follow\left(\{z_j\}\right) \wedge \forall j \in \{0,\ldots,k'-1\} : r \notin \bigcup_{\forall m' \in M} def(z_j,m')\right)\right)$$

Which is also equivalent, using the definition of *expected*, to:

$$\exists i \in I_e, \exists k' \in \mathbb{N}^+ : \left(i \in follow^{k'}(\{i_d\}) \wedge \exists i_f \in follow(\{i_d\}) : r \in expected(i_f,m)\right)$$

and also to:

$$\exists i \in I_e : i \in follow^+(\{i_d\}) \wedge r \in \bigcup_{\forall i_f \in follow(\{i_d\})} expected(i_f,m)$$

In this last expression, the condition $\exists i \in I_e : i \in follow^+(\{i_d\})$ is true if and only if $\exists i \in I_e : i \in follow(\{i_d\})$: if there exists an element which can be reached in a certain number of steps, there is for certain an intermediate element that can be reached in a single step.

The large union set is empty if there are no followers of i_d. Therefore $r \in \bigcup \ldots$implies that there is at least one follower. The condition $\exists i \in I_e : i \in follow^+(\{i_d\})$ is therefore superfluous, and the large initial expression can be ultimately reduced to:

$$r \in \bigcup_{\forall i_f \in follow(\{i_d\})} expected(i_f,m)$$

which proves the lemma. □

Lemma 5.3.3. *The expression*

$$\forall i \in I_e, \forall m \in M, \forall r \in R : r \in expected\,(i,m) \Rightarrow \Bigg(\forall i_d \in I_r : \Bigg(r \in \bigcup_{\forall m' \in M} def\,(i_d,m') \wedge \exists k \in \mathbb{N}^+ : \exists i_0,\ldots,i_k \in I_{pe} : \Bigg(i_0 = i_d \wedge i = i_k \wedge \forall j \in \{0,\ldots,k-1\} : i_{j+1} \in follow\left(\{i_j\}\right) \wedge \forall j \in \{1,\ldots,k-1\} : r \notin \bigcup_{\forall m' \in M} def\,(i_j,m') \Bigg) \Bigg) \Rightarrow r \in def\,(i_d,m) \Bigg)$$

is equivalent to

$$\forall m_1, m_2 \in M, m_1 \neq m_2, \forall i \in I_r, \forall i_f \in follow\,(\{i\}) : def\,(i,m_1) \cap expected'\,(i_f,m_2) = \varnothing$$

Proof. To simplify the notation, let us define a few auxiliary expressions as follows:

$$\begin{aligned} Y\,(i_d,i,r,m) &\Leftrightarrow \exists k \in \mathbb{N}^+ : \exists i_0,\ldots,i_k \in I_{pe} : \Bigg(i_0 = i_d \wedge i = i_k \wedge \forall j \in \{0,\ldots,k-1\} : i_{j+1} \in follow\left(\{i_j\}\right) \wedge \forall j \in \{1,\ldots,k-1\} : r \notin \bigcup_{\forall m' \in M} def\left(i_j,m'\right) \Bigg) \\ Z\,(i_d,r) &\Leftrightarrow r \in \bigcup_{\forall m' \in M} def\left(i_d,m'\right) \\ X\,(i_d,i,r,m) &\Leftrightarrow Y\,(i_d,i,r,m) \wedge Z\,(i_d,r) \end{aligned}$$

The first expression of the lemma can now be rewritten simply as:

$$\forall m \in M, \forall r \in R, \forall i \in I_e : r \in expected\,(i,m) \Rightarrow (\forall i_d \in I_r : (X\,(i_d,i,r,m) \Rightarrow r \in def\,(i_d,m)))$$

Rewriting the implications and moving the quantifiers, we also obtain the following equivalent expressions:

$$\forall m \in M, \forall r \in R, \forall i \in I_e : (\forall i_d \in I_r : (X\,(i_d,i,r,m) \Rightarrow r \in def\,(i_d,m))) \vee r \notin expected\,(i,m)$$

$$\forall m \in M, \forall r \in R, \forall i \in I_e : (\forall i_d \in I_r : (r \in def\,(i_d,m) \vee \neg X\,(i_d,i,r,m))) \vee r \notin expected\,(i,m)$$

$$\forall m \in M, \forall r \in R, \forall i \in I_e, \forall i_d \in I_r : (r \in def\,(i_d,m) \vee \neg X\,(i_d,i,r,m) \vee r \notin expected\,(i,m))$$

$$\forall m \in M, \forall r \in R, \forall i_d \in I_r, \forall i \in I_e :$$
$$(r \in def\,(i_d, m) \vee \neg Y\,(i_d, i, r, m) \vee \neg Z\,(i_d, r) \vee r \notin expected\,(i, m))$$

$$\forall m \in M, \forall r \in R, \forall i_d \in I_r, \forall i \in I_e :$$
$$(r \in def\,(i_d, m) \vee \neg Z\,(i_d, r) \vee (\neg Y\,(i_d, i, r, m) \vee \neg r \in expected\,(i, m)))$$

$$\forall m \in M, \forall r \in R, \forall i_d \in I_r :$$
$$(r \in def\,(i_d, m) \vee \neg Z\,(i_d, r) \vee \forall i \in I_e : (\neg Y\,(i_d, i, r, m) \vee \neg r \in expected\,(i, m)))$$

$$\forall m \in M, \forall r \in R, \forall i_d \in I_r :$$
$$(r \in def\,(i_d, m) \vee \neg Z\,(i_d, r) \vee \neg \exists i \in I_e : (Y\,(i_d, i, r, m) \wedge r \in expected\,(i, m)))$$

$$\forall m \in M, \forall r \in R, \forall i_d \in I_r :$$
$$(r \in def\,(i_d, m) \vee \neg\,(Z\,(i_d, r) \wedge \exists i \in I_e : (Y\,(i_d, i, r, m) \wedge r \in expected\,(i, m))))$$

$$\forall m \in M, \forall r \in R, \forall i_d \in I_r :$$
$$((Z\,(i_d, r) \wedge \exists i \in I_e : (Y\,(i_d, i, r, m) \wedge r \in expected\,(i, m))) \Rightarrow r \in def\,(i_d, m))$$

We can now use Lemma 5.3.2, obtaining:

$$\forall m \in M, \forall r \in R, \forall i_d \in I_r :$$
$$\left(\left(r \in \bigcup_{\forall m' \in M} def\,(i_d, m') \wedge r \in \bigcup_{\forall i_f \in follow(\{i_d\})} expected\,(i_f, m)\right) \Rightarrow r \in def\,(i_d, m)\right)$$

In this last expression, if $r \in def\,(i_d, m)$ the implication is automatically verified. However, if $r \in def\,(i_d, m')$, with $m \neq m'$, we know from Condition 5.2.4 that $r \notin def\,(i_d, m)$, which would invalidate the implication if $r \in \bigcup_{\forall i_f \in follow(\{i_d\})} expected\,(i_d, m)$ were also true. Consequently, this last condition must be false for the implication to be true. Hence:

$$\forall m_1, m_2 \in M, m_1 \neq m_2, \forall r \in R, \forall i_d \in I_r :$$
$$\left(r \notin def\,(i_d, m_1) \vee r \notin \bigcup_{\forall i_f \in follow(\{i_d\})} expected'\,(i_d, m_2)\right)$$

which is ultimately equivalent to:

$$\forall m_1, m_2 \in M, m_1 \neq m_2, \forall i \in I_r, \forall i_f \in follow\,(\{i\}) :$$
$$def\,(i, m_1) \cap expected\,(i_f, m_2) = \varnothing$$

□

Proposition 5.3.4. *Function body validity is equivalent to Condition 5.3.1:*

$$\forall m_1, m_2 \in M, m_1 \neq m_2, \forall i \in I_r, \forall i_f \in follow\,(\{i\}) :$$
$$def\,(i, m_1) \cap expected\,(i_f, m_2) = \varnothing$$

Proof. Immediate consequence of Lemma 5.3.3. □

Checking the simple Condition 5.3.1, we can easily verify that the function body is valid. To verify correctness, we only need to check whether the body is also sufficient. However, implementing the test described by Definition 5.2.13 is quite complicated, and a simpler test would be preferable. We will shortly see how to verify the same condition with less effort.

Condition 5.3.5. Each register that is expected in a given mode in the first instruction of the body must be supplied in that mode by the prologue:

$$\forall m \in M : expected\,(i_{first}, m) \subseteq def\,(enter, m)$$

Condition 5.3.5 must be verified in any case. We will now show that it also allows us to determine in an easy manner whether a function body is sufficient or not.

Proposition 5.3.6. *Condition 5.3.5 is equivalent to sufficiency:*

$$\forall m \in M : expected\,(i_{first}, m) \subseteq def\,(enter, m)$$

is equivalent to:

$$\forall m \in M, \forall r \in R, \forall k \in \mathbb{N}^+ : \big(i_0 = enter \wedge \big(\forall j \in \{0,\ldots,k-1\} : i_{j+1} \in follow\,\big(\{i_j\}\big)\big) \wedge r \in expected\,(i_k, m) \Rightarrow \exists j \in \{0,\ldots,k-1\} : r \in def\,(i_j, m)\big)$$

Proof. First of all, let us show that Condition 5.3.5 implies sufficiency. Ad absurdum, let Condition 5.3.5 be true but, for a particular $m \in M, r \in R, \langle i_0,\ldots,i_k\rangle \in I_{pe}^{k+1}$ we have that $r \in expected\,(i_k, m)$ and $\forall j \in \{0,\ldots,k-1\} : r \notin def\,(i_j, m)$. We know that $i_{first} \in follow\,(\{enter\})$ and, from the definition of *expected*, it is trivial to show that it must also be $r \in expected\,(i_{first}, m)$. There is an $i_u \in follow^q\,(\{i_k\})$, for some q, that uses r in mode m but, at the same time, $i_u \in follow^{q+k-1}\,\big(\{i_{first}\}\big)$ and there is no i_x between i_{first} and i_u for which $r \in def\,(i_x, m)$.

Since Condition 5.3.5 holds, however, we must also have $r \in def\,(enter, m)$. But then there *is* a j such that $r \in def\,(i_j, m)$, since i_0 is actually *enter*. We reach a contradiction, which shows that Condition 5.3.5 implies sufficiency.

Now let us show that sufficiency implies Condition 5.3.5. If the routine body is sufficient, and $r \in expected\,(i_{first}, m)$, then we know, since $i_{first} \in follow^1\,(\{enter\})$, that $\exists j \in \{0,\ldots,k-1\} : r \in def\,(i_j, m)$, where k is 1. But the only possible value for j is zero, which means that $r \in def\,(enter, m)$. □

Thanks to Proposition 5.3.6 and to Proposition 5.3.4, we have a straightforward way to test a routine body for several different conditions.

Proposition 5.3.7. *If a routine body respects the two following properties:*

$$\forall m \in M : expected\,(i_{first}, m) \subseteq def\,(enter, m)$$

$$\forall m_1, m_2 \in M, m_1 \neq m_2, \forall i \in I_r, \forall i_f \in follow(\{i\}) : def(i, m_1) \cap expected(i_f, m_2) = \varnothing$$

then it is correct, consistent, valid and sufficient.

Proof. As seen, the first property satisfies Condition 5.3.5, which in turn satisfies sufficiency by Proposition 5.3.6. The second property is equivalent to validity by Proposition 5.3.4. Finally, if the function body is both sufficient and valid, by Definition 5.2.15 it is also correct and, by Proposition 5.2.17, consistent. □

The following section will describe an algorithm capable of determining whether the properties listed in Proposition 5.3.7 are satisfied.

5.3.2 Implementation

The following algorithm implements all the tests described by Proposition 5.3.7, and can be run after the calculation of *exp* performed by Algorithm 1. Additionally, Condition 5.2.4 is also checked (limitedly to reachable instructions, as explained in the discussion following Proposition 5.2.17).

Algorithm 2 – Verification of routine body properties

procedure *verifyAll*()

Uses: Map *use* defined on $I_e \times M$
Uses: Map *def* defined on $I_p \times M$
Uses: Map *exp* defined on $I_e \times M$
Uses: Maps $follow_j$ defined on I_p for every $j = 0, \ldots, n_f - 1$

1: $reach \leftarrow \varnothing$
   {Stores I_r in *reach*}
2: *computeReachable*(*enter*)
3: **for all** $m_1 \in M$ **do**
4: **if** $exp[i_{first}, m_1] \nsubseteq def[i, m_1]$ **then** {Check sufficiency}
5: **error**
6: **end if**
7: **for all** $m_2 \in M, m_1 \neq m_2$ **do**
8: **for all** $i \in reach$ **do**
9: **if** $i \neq exit$ **then** {Check *def* and *use*}
10: **if** $def[i, m_1] \cap def[i, m_2] \neq \varnothing$ **then**
11: **error**
12: **end if**
13: **end if**
14: **if** $i \neq enter$ **then**
15: **if** $use[i, m_1] \cap use[i, m_2] \neq \varnothing$ **then**

```
16:              error
17:            end if
18:          end if
             {Check validity}
19:          if i ≠ exit then
20:            for all j such that 0 ≤ j < n_f do
21:              if follow_j[i] ≠ null then
22:                if def[i,m_1] ∩ exp[follow_j[i], m_2] ≠ ∅ then
23:                  error
24:                end if
25:              end if
26:            end for
27:          end if
28:        end for
29:      end for
30:  end for
```

procedure *computeReachable*(i)

Uses: Maps $follow_j$ defined on I_p for every $j = 0, \ldots, n_f - 1$

Uses: Set of instructions *reach*

Calculates: Adds to *reach* all the instr. reachable from i

```
1:  if i ≠ null then
2:    if i ∉ reach then
3:      reach ← reach ∪ {i}
4:      if i ≠ exit then
5:        for all j such that 0 ≤ j < n_f do
6:          computeReachable(follow_j[i])
7:        end for
8:      end if
9:    end if
10: end if
```

The time complexity of the above algorithm is actually lower than the complexity of Algorithm 1. The time required to calculate *reach* is $O(|I| \times n_f)$. The main checks require time $O\left(max\left(|M| \times |R|, |M|^2 \times |I| \times n_f\right)\right)$. Assuming that $|M|$ and $|R|$ are fixed and constant for a given architecture, and knowing that n_f can be potentially as large as $|I|$, the verification algorithm requires time $O\left(|I|^2\right)$. According to 5.3.7, if the algorithm terminates without error, it ensures that the routine body is correct, consistent, valid and sufficient. Additionally, it also ensures that the definitions of *def* and *use* are consistent, as far as reachable instructions are concerned.

5.4 Delay slots elimination

While the described algorithm is sufficiently general for many of the major machine architectures, a few microprocessors might require a particular treatment. The SPARC, for instance, uses a feature known as Delayed Control Transfer [Spa92]. The idea is that, when a branch or call is encountered, one additional instruction is executed before the control flow continues at the target of the control transfer instruction. The position occupied by such additional instruction is known as the "delay slot".

If delay slots are used, the peculiar behaviour of the control flow can no longer be described by the formal model that was used to describe the mode calculation algorithm. However, as we shall see in this section, it is still possible to obtain, from a routine description in which delay slots are used, a description of a routine body functionally equivalent in which delay slots are no longer present. The resulting transformed representation can then be used once again to perform the mode calculation and the sanity checks using the algorithms previously described.

In the following discussion, it will be assumed that some of the instructions present in a routine body can make use of delay slots. The description will be rather generic, and not tied to any particular microprocessor, but the SPARC will be sometimes used for illustration. For the more obscure details of the Delayed Control Transfer mechanism, it may be useful to refer to the SPARC manuals.

5.4.1 A model for delay slots

In order to simplify the discussion, it will be assumed that there are only two possible followers for every instruction ($n_f = 2$), and indirect jumps are not used. Extending the discussion to multiple followers is fairly straightforward. In order to describe delay slots and proceed with the transformation of the routine body, a formal notation will be used to categorize instructions in groups, in order to distinguish the various effects of the delay slots. Similarly, the presence of a delay slot for every instruction will be described.

Definition 5.4.1. The set of instruction types is $T = \{simple, cond, uncond, call\}$. The set of instruction delays is $D = \{nodelay, delayed, annul\}$. An element from each of the two sets is assigned to each instruction using the two functions $type : I \to T$ and $delay : I \to D$.

The meaning of the elements of T is the following:

- if $type(i) = simple$ the instruction is a plain instruction that does not alter the control flow
- if $type(i) = cond$ the instruction is a conditional branch; execution might continue to the branch target or continue undisturbed
- if $type(i) = uncond$ the instruction is an unconditional branch; execution continues to the branch target
- if $type(i) = call$ the instruction is a call instruction; execution continues to the subroutine and will return later

If the compiler determines that a call to subroutine never returns, the corresponding call instruction can be treated, for mode calculation purposes, as an unconditional branch to *null* which uses certain registers in certain modes. The elements of D have the following meaning:

- if $delay(i) = nodelay$ the instruction does not use a delay slot
- if $delay(i) = delayed$ the instruction is a control transfer instruction that uses a delay slot
- if $delay(i) = annul$ the instruction has a delay slot, but, as described below, the instruction in the delay slot is sometimes ignored

The condition *annul* is used to describe the case in which the instruction in the delay slot is executed if a conditional branch is taken, but ignored if the conditional branch is not taken. This feature, present in the SPARC for instance, can be used to implement simple if-then-else sequences with just a single branch (more details later). It should be noted that not all of the combinations are, in general, meaningful, and it could be useful to detect the illegal combinations with an explicit verification, as it will be explained in the following section.

The way in which the functions $follow_j$ are used also depends on the instruction type, as follows:

- if $type(i) \in \{simple, call\}$ then $follow_0(i) = succ(i)$ and $follow_1(i) = null$;
- if $type(i) = cond$ then $follow_0(i) = succ(i)$ and $follow_1(i) = i_t$, where i_t is the representation of the instruction which is the target of the conditional branch (possibly *exit*);
- if $type(i) = uncond$ then $follow_0(i) = null$ and $follow_1(i) = i_t$, where i_t is the representation of the instruction which is the target of the unconditional branch (possibly *exit*).

The values of $follow_j$, in the case of branches, are chosen so that $follow_0$ is the next instruction if the branch is not taken, and $follow_1$ is the next instruction if the branch is taken. If delay slots are used, the the functions def, use, $follow_0$ and $follow_1$ do not describe the effects on registers and control flow of the various instructions in a manner that is of any use to the mode calculation algorithm, since the branch targets are not followed immediately but in a deferred way. In order to determine what exactly happens to registers, a custom algorithm will create, from the functions def, use, $follow_0$ and $follow_1$, the functions def', use', $follow'_0$ and $follow'_1$, eliminating along the way the effect of the delay slots while preserving the overall effect on the registers of the original program, and of course the same sequence of instructions.

5.4.2 Conditions on delay slots

Not every possible combination for $type$, $follow_j$ and $delay$ is legal. Due to the particular behaviour of the delay slot mechanism, which we are trying to model formally, certain restrictions or difficulties will need to be analysed with a bit of care.

5.4.2.1 Delay slots as branch targets

As previously mentioned, the instruction in the delay slot is actually executed before the new value of the program counter, specified by the branch instruction, is actually followed. A particular issue arises when the control flow can enter the instruction sequence directly with the instruction in the delay slot. That could happen if a branch instruction, located somewhere in the same routine body, has as its target the instruction in the delay slot of some other instruction.

Figure 5.4.1 shows the two, and unrelated, possible execution paths that can traverse the instruction in the delay slot. The first possibility involves i_o and i_1, then i_2 is executed before the branch is taken, and execution then continues with i_x. The second possibility involves a jump from some other location to i_2; execution then continues with i_3 and the following instructions. The presence of two distinct and unrelated paths, however, presents a potential problems regarding the calculation of register contents before the execution of instruction i_2. The way in which the various registers are expected before the execution of i_2 would depend on the alternative followed, and it would be necessary to maintain the two values of a function like *expected* in the two cases.

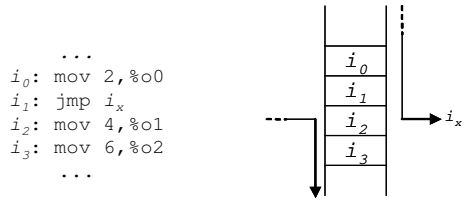


Figure 5.4.1: Delay slot used as branch target

In the case of the SPARC, one of the internal registers, named nPC, contains at any given moment the value of the program counter at which the execution will continue after the current instruction. It is therefore possible to establish which of the two alternatives is being followed at any given moment by verifying the current value of nPC. However, it is still necessary to maintain two possible entries for the instruction in the delay slot.

The issue is not present if the instruction in the delay slot is never a target of a branch within the body. In that case, the instruction in the delay slot is always executed as part of the branch, and there is no ambiguity. In practice, if an instruction in a delay slot is a target of a branch, it means that the compiler managed to perform a rather complex optimisation, transferring one of the instructions that precede logically the branch in the delay slot, while at the same time discovering that the same instruction can be used following a completely different execution path, consolidating the two in a single machine instruction and calculating correctly all the possible effects of such an instruction combination.

Unsurprisingly, GCC appears not to perform this peculiar kind of optimisation, at least according to the tests performed in the context of this research. The delay slot elimination algorithm that will be described later relies on delay slots to be used in a single execution path. No delay slots, therefore, can be the target of a branch instruction. In formal terms, the following condition must be verified as a precondition to the algorithm:

Condition 5.4.2. No delay slot can be the target of a branch instruction:

$$\forall i_0 \in I : delay(i_0) \neq nodelay \Rightarrow (\nexists i_1 \in I : follow_1(i_1) = succ(i_0))$$

5.4.2.2 Control Transfer Instructions in delay slots

One additional issue related to delay slots is the microprocessor behaviour in case a control transfer instruction (a branch or a call) is present in the delay slot of another instruction. Such a combination is referred to in the SPARC architecture manual as a "DCTI couple" (Delayed Control Transfer Instruction couple), and can cause the control flow to follow a quite complex path depending on the use of annulled delays and whether the branches are taken or not. While the various cases are documented in the specific case of the SPARC (with a behaviour sometimes listed as "unpredictable"), there is very little generality to it, since the specific behaviour depends on the internal implementation of the specific microprocessor architecture considered. Furthermore, the complexity of the various cases make the DCTI couple practically unusable in automatically generated code. In order to maintain the generality of the delay slot elimination algorithm and to simplify the description, it will be assumed that only simple instructions can appear in delay slots. The property is verified in the code generated by GCC, since only non-delayed instructions are eligible for use in a delay slot. Formally:

Condition 5.4.3. Only simple instructions can be used in delay slots.

$$\forall i \in I, succ(i) \neq exit : delay(i) \neq nodelay \Rightarrow type(succ(i)) = simple$$

5.4.2.3 Last instruction in the routine body

To complete the cases, it will also be required that the last instruction in the body is not a delayed instruction. That would otherwise cause the first instruction following the routine body (which is unknown in this model) to be executed as part of the body. That is, the instruction represented by *exit* would be executed before the branch/call instruction. The following condition, therefore, must also be verified:

Condition 5.4.4. The last instruction of the body must not be delayed:

$$delay\left(succ^{-1}(exit)\right) = nodelay$$

5.4.2.4 Possible uses of delay slots

Finally, we will require that only branches or calls can have delay slots, and that calls do not use annulled delays.

Condition 5.4.5. A simple instruction cannot be delayed. Calls cannot use annulled delays.

$$\forall i \in I : type(i) = simple \Rightarrow delay(i) = nodelay$$

$$\forall i \in I : type(i) = call \Rightarrow delay(i) \neq annul$$

As an additional note, we will assume that the only side effect of a branch or call instruction that is actually delayed is the late modification of the value of the program counter. Tests and possible side effects on registers are still assumed to take place when the branch instruction is encountered

during execution. Consequently, for instance, if the branch tests the state of a certain register and the instruction in the delay slot alters the same register, the latter instruction is assumed not to have any effect on the branch decision.

5.4.3 A few examples

Let us see how the representation just introduced can be used to describe some common cases involving the use of delay slots.

5.4.3.1 Unconditional branch

In the example represented in Figure 5.4.2, instruction i_1 is a delayed unconditional branch. When the execution flow reaches i_1, the address of the branch target is just fetched, but not immediately followed. The control flow reaches then i_2, the instruction in the delay slot. Only after the execution of i_2 is complete, the branch target is followed. In this case, $type(i_1) = uncond$ and $delay(i_1) = delayed$. Since i_1 is an unconditional branch, initially $follow_0(i_1) = null$ and $follow_1(i_1) = i_x$. The instruction in the delay slot, i_2, is a regular instruction, so $type(i_2) = simple$, $delay(i_2) = nodelay$, $follow_0(i_2) = i_3$ and $follow_1(i_2) = null$.

The mode calculation algorithm cannot be directly applied to this code fragment, because the functions $follow_j$ do not reflect the real behaviour of the control flow. However, it is possible to describe the real behaviour defining the functions $follow'_j$ as follows:

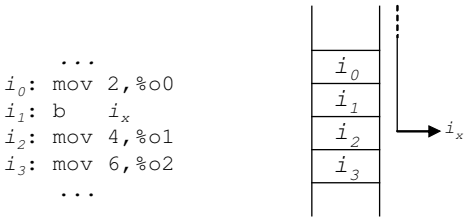


Figure 5.4.2: Unconditional branch

- $follow'_1(i_1) = null$, because the branch does not have immediate effect
- $follow'_0(i_1) = i_2$, since execution continues with the instruction in the delay slot
- $follow'_1(i_2) = i_x$, describing that, after the delay slot, execution continues at the branch target
- $follow'_0(i_2) = null$, since the branch is unconditional

Using $follow'_j$ in place of $follow_j$, it is now possible to proceed with the usual mode calculation. If a conditional branch were used, rather than an unconditional branch, the only difference would have been $follow'_0(i_2) = i_3$, in order to describe the alternative in case the branch is not taken.

5.4.3.2 Annulled, branch always

Let us consider the case of an unconditional branch (branch always) in which the instruction in the delay slot is annulled.[3] The instruction in the delay slot is fetched by the microprocessor but not executed, and execution continues with the instruction specified as the branch target. A delayed "branch always" instruction, flagged as "annulled", is therefore mostly equivalent to a non-delayed unconditional branch, but the control flow can actually reach the instruction in the delay slot, which behaves as a no-operation. Consequently, $follow'_j$ will be calculated in exactly the same way as the case above. However, $use'(i_2, m) = \varnothing$ and $def'(i_2, m) = \varnothing$, to indicate that the instruction in the delay slot has no effect.

5.4.3.3 Annulled, conditional branch

According to the SPARC specifications, in the case of conditional delayed branches in which the delay slot is annulled, the instruction in the delay slot is ignored if the branch is not taken. For instance, in the example represented in Figure 5.4.3, the control flow reaches instruction i_1, which is an "annulled" delayed conditional branch.

The branch target, i_x, is discovered by the microprocessor but, since the branch is delayed, control is passed to the following instruction. Instruction i_2 is now fetched but, before execution, the branch test is checked. If the branch is taken, then i_2 is executed normally and execution continues with i_x. If, conversely, the branch is not taken, then i_2 is ignored, and control passes to the following instruction, i_3. Annulled delay slots, when used with conditional branches, can be used to create very compact implementations of small if-then-else control structures, for instance when i_2 is one of the two alternatives to be executed as a result of the test. Although the situation is a little more complex than the previous cases, rearranging $follow_j$ is still fairly easy, following the following scheme:

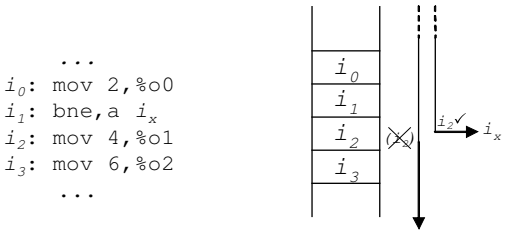


Figure 5.4.3: Conditional branch, annul

- $follow'_0(i_1) = i_2$, since execution continues with the instruction in the delay slot
- $follow'_1(i_1) = i_3$, because, if the delay slot is annulled, execution virtually continues with the following instruction[4]

[3] The SPARC also allows a "Branch Never" unconditional instruction, which does nothing. If such an instruction is flagged as "annulled", the instruction in the delay slot is always ignored, and the execution continues with the following instruction. That instruction can be easily transformed, for the purposes of the mode analysis, in a non-delayed unconditional branch (always) instruction that skips the instruction in the delay slot. This case is not considered in the delay slot elimination algorithm, described later, since the usefulness of such an instruction is extremely limited.

[4] To be entirely accurate, $follow'_j$ should be set to i_2 in both cases, and then, using some special mechanism, the fact that the instruction in the delay slot is sometimes ignored should be somehow represented. However, the desired result of the rearrangement is the ability to calculate the expected modes before every instruction, using the mode calculation algorithm. In this respect, the proposed calculation of $follow'_j$ is entirely consistent with the intended result.

- $follow'_1(i_2) = i_x$, after the delay slot, execution continues at the branch target
- $follow'_0(i_2) = null$, if the instruction was executed, then the branch is certainly taken

5.4.4 Delayed calls

So far, the cases of branches and annulled branches have been described. As seen, the functions $follow'_j$ can be easily constructed from the corresponding $follow_j$ so that the delayed execution of branches is taken into account. In the case of calls, however, the situation is considerably more complex.

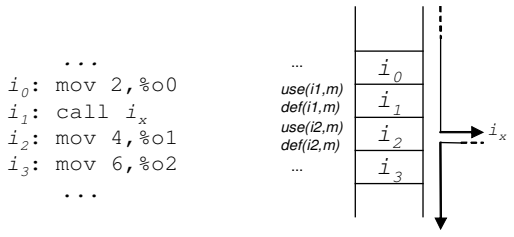


Figure 5.4.4: Delayed call

Section 5.2.8 explained the reasons why a call instruction can be considered to have an effect similar to that of a simple instruction, as far as the mode calculation is concerned. The idea is that a call invocation "uses" some registers (as parameters), performs some operations, and returns to the instruction following the call, after "defining" some registers (either return values or overwriting them with some unknown values). The overall effect of the call on registers is therefore similar to that of a single instruction.

If delay slots are involved, however, execution actually returns to the instruction following the one that is in the delay slot. Furthermore, since the jump to the called subroutine is performed in a delayed fashion, the call instruction itself appears, during execution, not to have any immediate effect. The diagram in Figure 5.4.4 will be useful to explain what exactly happens.

1. The control flow reaches i_0, which uses certain registers, defines other registers and passes control to i_1.
2. The microprocessor executes instruction i_1, which is a delayed call. The microprocessor detects that the target is i_x, but wants to execute one further instruction before jumping. Crucially, at this stage no register is actually altered. Control passes to i_2.
3. Instruction i_2 is executed normally, affecting the registers according to $def(i_2,m)$ and $use(i_2,m)$ for every m. After the execution of i_2, the microprocessor jumps to i_x. The subroutine is executed, causing the registers to be modified according to $def(i_1,m)$ and $use(i_1,m)$ for every m, the callee returns and control is passed to instruction i_3.
4. Instruction i_3 is executed normally.

To calculate def', use' and the $follow'_j$, in order to obtain a description of the effect that an equivalent program would have if delay slots were not used, it is therefore necessary to set $def'(i_2,m)$

and $use'(i_2, m)$ to the combined effect that i_2 and i_1 would have, in that order. The equivalent result is shown in Figure 5.4.5. In particular, it should be noted that some inconsistencies in *def* and *use* might be concealed by the effect of the combination, and a restricted form of the conditions previously mentioned must be explicitly verified prior to the combination, as explained in detail later.

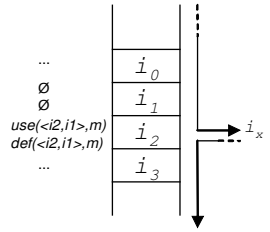


Figure 5.4.5: Equivalent representation for a delayed call

5.4.5 Combining instructions

As emerged in the previous section, it is sometimes necessary to merge the effect of two distinct instructions in order to obtain an equivalent effect for a single instruction. In our model, that means calculating the values that *def* and *use* would have for such a combined instruction. In particular, let us assume that we have a sequence of instructions i_0, i_1 and i_2, in which i_1 is the only follower of i_0, and i_2 the only follower of i_1.

The values of $def(i_0, m)$, $use(i_0, m)$, $def(i_1, m)$ and $use(i_1, m)$ are already known for every $m \in M$. Intuitively, a register is used in mode m by the pair $\langle i_0, i_1 \rangle$ if it is used in that mode by i_0, or if it is not defined by i_0 and yet it is used by i_1. Similarly, a register is defined in mode m by the pair $\langle i_0, i_1 \rangle$ if it is defined by i_1, or if it is defined by i_0 in that mode but it is not defined in some other mode by i_1. Formally:

$$\forall m \in M, \forall r \in R: \begin{cases} r \in def(\langle i_0, i_1 \rangle, m) \iff r \in def(i_1, m) \vee \left(r \in def(i_0, m) \wedge r \notin \bigcup_{\forall m' \in M} def(i_1, m') \right) \\ r \in use(\langle i_0, i_1 \rangle, m) \iff r \in use(i_0, m) \vee \left(r \in use(i_1, m) \wedge r \notin \bigcup_{\forall m' \in M} def(i_0, m') \right) \end{cases}$$

The definition above is quite intuitive, but it may be worth verifying that the mode calculation algorithm actually still works as expected when the two separate instructions are replaced by the combined one. In particular, the value of *expected* before the execution of the combined instruction must be identical to the value that *expected* would assume before i_0 if the two instructions are separate. In other terms:

Proposition 5.4.6. *If i_0 and i_1 are replaced with a new instruction i_{01}, having the effect of the pair $\langle i_0, i_1 \rangle$, then*

$$\forall m \in M : expected(i_{01}, m) = expected(i_0, m)$$

Proof. Using the definition of operator T (Def. 5.2.19), we know that

$$\forall m \in M, \forall r \in R : r \in expected\,(i_0, m) \Leftrightarrow r \in use\,(i_0, m) \vee \left(r \notin \bigcup_{\forall m' \in M} def\,(i_0, m') \wedge \right.$$
$$\left. r \in use\,(i_1, m) \vee \left(r \notin \bigcup_{\forall m' \in M} def\,(i_1, m') \wedge r \in expected\,(i_2, m) \right) \right)$$

and the new value of *expected* is:

$$\forall m \in M, \forall r \in R : r \in expected\,(i_{01}, m) \Leftrightarrow r \in use\,(i_{01}, m) \vee \left(r \notin \bigcup_{\forall m' \in M} def\,(i_{01}, m') \wedge r \in expected\,(i_2, m) \right)$$

where $use\,(i_{01}, m) = use\,(\langle i_0, i_1 \rangle, m)$ and $def\,(i_{01}, m) = def\,(\langle i_0, i_1 \rangle, m)$.

The above definition of $expected\,(i_0, m)$ will be manipulated in order to obtain an equivalent expression. In order to simplify the notation, some symbols will be used to rewrite the expressions, with the following meaning:

$$\mathcal{A} : r \in use\,(i_0, m) \quad \mathcal{B} : r \notin \bigcup_{\forall m' \in M} def\,(i_0, m') \quad \mathcal{C} : r \in use\,(i_1, m)$$
$$\mathcal{D} : r \notin \bigcup_{\forall m' \in M} def\,(i_1, m') \quad \mathcal{E} : r \in expected\,(i_2, m)$$

The expression for $expected\,(i_0, m)$ can now be rewritten as follows:

$$\forall m \in M, \forall r \in R : r \in expected\,(i_0, m) \Leftrightarrow$$

$$\mathcal{A} \vee (\mathcal{B} \wedge (\mathcal{C} \vee (\mathcal{D} \wedge \mathcal{E}))) \Leftrightarrow$$

$$(\mathcal{A} \vee \mathcal{B}) \wedge (\mathcal{A} \vee \mathcal{C} \vee (\mathcal{D} \wedge \mathcal{E})) \Leftrightarrow$$

$$(\mathcal{A} \vee \mathcal{B}) \wedge (\mathcal{A} \vee \mathcal{C} \vee \mathcal{D}) \wedge (\mathcal{A} \vee \mathcal{C} \vee \mathcal{E}) \Leftrightarrow$$

$$[((\mathcal{A} \vee \mathcal{B}) \wedge (\mathcal{A} \vee \mathcal{C} \vee \mathcal{D})) \wedge (\mathcal{A} \vee \mathcal{C})] \vee [((\mathcal{A} \vee \mathcal{B}) \wedge (\mathcal{A} \vee \mathcal{C} \vee \mathcal{D})) \wedge \mathcal{E}] \Leftrightarrow$$

$$[(\mathcal{A} \vee \mathcal{B}) \wedge (\mathcal{A} \vee \mathcal{C})] \vee [(\mathcal{A} \vee (\mathcal{B} \wedge (\mathcal{C} \vee \mathcal{D}))) \wedge \mathcal{E}] \Leftrightarrow$$

$$[(\mathcal{A} \vee (\mathcal{B} \wedge \mathcal{C}))] \vee [((\mathcal{A} \vee (\mathcal{B} \wedge \mathcal{C})) \vee (\mathcal{B} \wedge \mathcal{D})) \wedge \mathcal{E}] \Leftrightarrow$$

$$[(\mathcal{A} \vee (\mathcal{B} \wedge \mathcal{C}))] \vee [((\mathcal{A} \vee (\mathcal{B} \wedge \mathcal{C})) \wedge \mathcal{E}) \vee ((\mathcal{B} \wedge \mathcal{D}) \wedge \mathcal{E})] \Leftrightarrow$$

$$[(\mathcal{A} \vee (\mathcal{B} \wedge \mathcal{C})) \vee ((\mathcal{A} \vee (\mathcal{B} \wedge \mathcal{C})) \wedge \mathcal{E})] \vee [(\mathcal{B} \wedge \mathcal{D}) \wedge \mathcal{E}] \Leftrightarrow$$

$$[(\mathcal{A} \vee (\mathcal{B} \wedge \mathcal{C}))] \vee [(\mathcal{B} \wedge \mathcal{D}) \wedge \mathcal{E}] \Leftrightarrow$$

$$\left[r \in use\,(i_0,m) \vee \left(r \notin \bigcup_{\forall m' \in M} def\,(i_0,m') \wedge r \in use\,(i_1,m)\right)\right] \vee$$
$$\left[\left(r \notin \bigcup_{\forall m' \in M} def\,(i_0,m') \wedge r \notin \bigcup_{\forall m' \in M} def\,(i_1,m')\right) \wedge r \in expected\,(i_2,m)\right]$$

A similar transformation is now performed on $expected\,(i_{01},m)$. Expanding $use\,(i_{01},m)$ and $def\,(i_{01},m')$:

$$\forall m \in M, \forall r \in R : r \in expected\,(i_{01},m) \Leftrightarrow$$

$$r \in use\,(i_{01},m) \vee \left(r \notin \bigcup_{\forall m' \in M} def\,(i_{01},m') \wedge r \in expected\,(i_2,m)\right) \Leftrightarrow$$

$$\left[r \in use\,(i_0,m) \vee \left(r \in use\,(i_1,m) \wedge r \notin \bigcup_{\forall m' \in M} def\,(i_0,m')\right)\right] \vee$$
$$\left[\forall m'' \in M : \neg\left(r \in def\,(i_1,m'') \vee \left(r \in def\,(i_0,m'') \wedge r \notin \bigcup_{\forall m' \in M} def\,(i_1,m')\right)\right) \wedge r \in expected\,(i_2,m)\right]$$

This last expression is quite similar to the previous expansion of $expected\,(i_0,m)$. Some additional transformations can now be performed on this subexpression:

$$\forall m'' \in M : \neg\left(r \in def\,(i_1,m'') \vee \left(r \in def\,(i_0,m'') \wedge r \notin \bigcup_{\forall m' \in M} def\,(i_1,m')\right)\right) \Longleftrightarrow$$

$$\forall m'' \in M : \left(r \notin def\,(i_1,m'') \wedge \left(r \notin def\,(i_0,m'') \vee r \in \bigcup_{\forall m' \in M} def\,(i_1,m')\right)\right) \Longleftrightarrow$$

$$\forall m'' \in M : \left(r \notin def\,(i_1,m'')\right) \wedge \forall m'' \in M : \left(r \notin def\,(i_0,m'') \vee r \in \bigcup_{\forall m' \in M} def\,(i_1,m')\right) \Longleftrightarrow$$

$$r \notin \bigcup_{\forall m' \in M} def\,(i_1,m') \wedge \left(r \notin \bigcup_{\forall m' \in M} def\,(i_0,m') \vee r \in \bigcup_{\forall m' \in M} def\,(i_1,m')\right) \Longleftrightarrow$$

$$r \notin \bigcup_{\forall m' \in M} def\,(i_1, m') \wedge r \notin \bigcup_{\forall m' \in M} def\,(i_0, m')$$

Which shows that $\forall m \in M : expected\,(i_{01}, m) = expected\,(i_0, m)$. □

The previous proposition shows that the definitions of *def* and *use* on instruction pairs is still consistent with the mode calculation. In other words, if an instruction pair is replaced with a single combined instruction, it is still possible to run the mode calculation algorithm obtaining the same results. In Section 5.3, however, the function *expected* was also used to perform a number of sanity checks on the original definitions of *def* and *use*. Combining two instructions into one, we might lose some information, and miss some inconsistencies.

In detail, the basic idea is to transform the definitions of *def*, *use* and $follow_j$ into def', use' and $follow'_j$, so that the effect of the delay slots is incorporated into the new functions. Then the mode calculation algorithm is run using the latter transformed functions instead of the original ones. Finally, the sanity checks are performed, using the result of the previous algorithm.

In the case in which two instructions are combined, it is necessary to make sure that the sanity checks, performed in the last stage on the new instruction i_{01}, are at least as stringent as the tests which would have been performed on the original instructions. If there is a chance of some error conditions going undetected, then it is necessary to perform additional tests while merging the two instructions, as will be explained more in detail in the following section.

5.4.6 Sanity checks for combined instructions

There are three essential sanity checks that are performed by Algorithm 2: validity and sufficiency of the routine body, plus consistency of *def* and *use*, according to Condition 5.2.4. The equivalent conditions in the case of combined instructions will now be discussed.

5.4.6.1 Validity

The intuitive idea behind validity asserts that every register which is expected in a certain mode must be set exactly in that mode by the first preceding definition along every possible execution path. In Proposition 5.3.4, an alternate condition was given for validity:

$$\forall m_1, m_2 \in M, m_1 \neq m_2, \forall i \in I_r, \forall i_f \in follow\,(\{i\}) : \\ def\,(i, m_1) \cap expected\,(i_f, m_2) = \varnothing$$

Assuming that the instruction i_0 of our example is reachable, checking validity on i_0 and i_1 means:

$$\forall m_1, m_2 \in M, m_1 \neq m_2 : \begin{cases} def\,(i_1, m_1) \cap expected\,(i_2, m_2) = \varnothing \\ def\,(i_0, m_1) \cap expected\,(i_1, m_2) = \varnothing \end{cases}$$

that is:

$$\forall m_1, m_2 \in M, m_1 \neq m_2 : \begin{cases} def\,(i_1, m_1) \cap expected\,(i_2, m_2) = \varnothing \\ def\,(i_0, m_1) \cap \left(use\,(i_1, m_2) \cup \left(expected\,(i_2, m_2) \setminus \bigcup_{\forall m' \in M} def\,(i_1, m') \right) \right) = \varnothing \end{cases} \tag{5.4.1}$$

Using the same test described in Section 5.3.4 for the combined instruction, the test is:

$$\forall m_1, m_2 \in M, m_1 \neq m_2 : def\,(i_{01}, m_1) \cap expected\,(i_2, m_2) = \varnothing \tag{5.4.2}$$

In order to check whether the new test is sufficiently strong, we would like to verify whether the test 5.4.2 is equivalent to the test 5.4.1. Expanding the expression in test 5.4.2:

$$\forall m_1, m_2 \in M, m_1 \neq m_2 : \left(def\,(i_1, m_1) \cup \left(def\,(i_0, m_1) \setminus \bigcup_{\forall m' \in M} def\,(i_1, m') \right) \right) \cap expected\,(i_2, m_2) = \varnothing$$

$$\forall m_1, m_2 \in M, m_1 \neq m_2 : (def\,(i_1, m_1) \cap expected\,(i_2, m_2)) \cup \left(\left(def\,(i_0, m_1) \setminus \bigcup_{\forall m' \in M} def\,(i_1, m') \right) \cap expected\,(i_2, m_2) \right) = \varnothing$$

$$\forall m_1, m_2 \in M, m_1 \neq m_2 : \begin{cases} def\,(i_1, m_1) \cap expected\,(i_2, m_2) = \varnothing \\ \left(def\,(i_0, m_1) \setminus \bigcup_{\forall m' \in M} def\,(i_1, m') \right) \cap expected\,(i_2, m_2) = \varnothing \end{cases} \tag{5.4.3}$$

The first lines of 5.4.3 and 5.4.1 are identical, but it is still necessary to compare the second lines of the two expressions. The second line of expression 5.4.3 can be written as:

$$\forall m_1, m_2 \in M, m_1 \neq m_2, \forall r \in R : \neg \left(r \in expected\,(i_2, m_2) \wedge \left(r \in def\,(i_0, m_1) \wedge r \notin \bigcup_{\forall m' \in M} def\,(i_1, m') \right) \right)$$

$$\forall m_1, m_2 \in M, m_1 \neq m_2, \forall r \in R : \left(r \notin def\,(i_0, m_1) \vee \left(r \in \bigcup_{\forall m' \in M} def\,(i_1, m') \vee r \notin expected\,(i_2, m_2) \right) \right) \tag{5.4.4}$$

The second line of 5.4.1 can be written as

$$\forall m_1, m_2 \in M, m_1 \neq m_2, \forall r \in R : \neg \left(r \in def\,(i_0, m_1) \wedge \left(r \in use\,(i_1, m_2) \vee \left(r \notin \bigcup_{\forall m' \in M} def\,(i_1, m') \wedge r \in expected\,(i_2, m_2) \right) \right) \right)$$

$$\forall m_1, m_2 \in M, m_1 \neq m_2, \forall r \in R :$$
$$\left(r \notin def\,(i_0, m_1) \vee \left(\left(r \in \bigcup_{\forall m' \in M} def\,(i_1, m') \vee r \notin expected\,(i_2, m_2) \right) \wedge r \notin use\,(i_1, m_2) \right) \right) \tag{5.4.5}$$

Written in this form, the main subexpression of 5.4.4 has the form $\mathcal{F} \vee \mathcal{G}$, while the main subexpression of 5.4.5 has the form $\mathcal{F} \vee (\mathcal{G} \wedge \mathcal{H})$, where

$$\mathcal{F} : r \notin def\,(i_0, m_1) \quad \mathcal{G} : \left(r \in \bigcup_{\forall m' \in M} def\,(i_1, m') \vee r \notin expected\,(i_2, m_2) \right) \quad \mathcal{H} : r \notin use\,(i_1, m_2)$$

The two subexpressions in 5.4.4 and in 5.4.5 are equivalent for all the truth values of $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{H}$, except in the case $\mathcal{F} = false$, $\mathcal{G} = true$, and $\mathcal{H} = false$, which satisfies the subexpression in 5.4.4 but not the one in 5.4.5. In other words, if $\neg\mathcal{F} \wedge \mathcal{G} \wedge \neg\mathcal{H}$ then validity in the case of separate instructions is not satisfied, but validity in the case of combined instructions is (incorrectly) satisfied. The validity test on the combined instructions is consequently not as strong as the original validity test for the separated instructions. On top of condition 5.4.2 it is necessary to also verify that $\neg(\neg\mathcal{F} \wedge \mathcal{G} \wedge \neg\mathcal{H})$, that is $\mathcal{F} \vee \neg\mathcal{G} \vee \mathcal{H}$:

$$\mathcal{F} \vee (\mathcal{G} \wedge \mathcal{H}) \Leftrightarrow (\mathcal{F} \vee \mathcal{G}) \wedge (\mathcal{F} \vee \neg\mathcal{G} \vee \mathcal{H})$$

and, since $\mathcal{F} \vee \mathcal{G}$ also implies $\mathcal{F} \vee \mathcal{G} \vee \mathcal{H}$:

$$\mathcal{F} \vee (\mathcal{G} \wedge \mathcal{H}) \Leftrightarrow (\mathcal{F} \vee \mathcal{G}) \wedge (\mathcal{F} \vee \mathcal{G} \vee \mathcal{H}) \wedge (\mathcal{F} \vee \neg\mathcal{G} \vee \mathcal{H})$$

$$\mathcal{F} \vee (\mathcal{G} \wedge \mathcal{H}) \Leftrightarrow (\mathcal{F} \vee \mathcal{G}) \wedge (\mathcal{F} \vee \mathcal{H})$$

Summarising, when instructions are combined, in addition to the validity test in 5.4.2 the following test must also be verified in order to guarantee the validity of the original routine:

$$\forall m_1, m_2 \in M, m_1 \neq m_2, \forall r \in R : r \notin def\,(i_0, m_1) \vee r \notin use\,(i_1, m_2)$$

which is also equivalent to:

$$\forall m_1, m_2 \in M, m_1 \neq m_2, \nexists r \in R : r \in def\,(i_0, m_1) \wedge r \in use\,(i_1, m_2)$$

and finally to:

$$\forall m_1, m_2 \in M, m_1 \neq m_2 : def\,(i_0, m_1) \cap use\,(i_1, m_2) = \varnothing \tag{5.4.6}$$

5.4.6.2 Sufficiency

The definition of sufficiency (Def. 5.2.13) requires each register expected in a certain mode at any reachable instruction to be matched by a preceding definition of the same register in the same mode along every execution path. If an execution path traverses i_0 and i_1 in the original routine

body, it will also traverse the new instruction i_{01} in the modified routine body. The definition of sufficiency was:

$$\forall m \in M, \forall r \in R, \forall k \in \mathbb{N}^+ : \left(i'_0 = enter \wedge \left(\forall j \in \{0,\ldots,k-1\} : i'_{j+1} \in follow\left(\left\{i'_j\right\}\right)\right) \wedge r \in expected\left(i'_k, m\right) \Rightarrow \exists j \in \{0,\ldots,k-1\} : r \in def\left(i'_j, m\right)\right)$$

We can consider various cases, to compare the meaning of sufficiency for the original routine body with sufficiency on the modified routine. It is necessary to verify that we can find an element i'_j of the sequence for which $r \in def\left(i'_j, m\right)$, whatever the chosen sequence is.

1. If, for a certain i'_k, it is $r \in expected\left(i'_k, m\right)$ and no element of the subsequence $\left\langle i'_0, \ldots, i'_{k-1}\right\rangle$ is equal to i_0, then verifying sufficiency for that sequence on the transformed code is the same as doing it for the original routine.

2. If the subsequence $\left\langle i'_0, \ldots, i'_{k-1}\right\rangle$ includes at some point i_0 and i_1, one after the other, then the original definition says that $\exists j \in \{0,\ldots,k-1\} : r \in def\left(i'_j, m\right)$. Let $W = \left\{i'_j | j = 0,\ldots,k-1\right\} \setminus \{i_0, i_1\}$. If $\exists i_x \in W : r \in def\left(i_x, m\right)$, then i_x is part of the modified routine, and part of a similar sequence in which i_0 and i_1 are replaced by i_{01}. If, on the other hand, it is $\nexists i_x \in W : r \in def\left(i_x, m\right)$, then sufficiency implies that in the original routine $r \in def\left(i_0, m\right) \vee r \in def\left(i_1, m\right)$, while in the modified routine $r \in def\left(i_{01}, m\right)$. We need to make sure that the latter implies the former. Expanding $r \in def\left(i_{01}, m\right)$:

$$r \in def\left(i_{01}, m\right) \iff r \in def\left(i_1, m\right) \vee \left(r \in def\left(i_0, m\right) \wedge r \notin \bigcup_{\forall m' \in M} def\left(i_1, m'\right)\right)$$

   which implies $r \in def\left(i_0, m\right) \vee r \in def\left(i_1, m\right)$. Therefore, verifying the test for a certain i'_k along $\left\langle i'_0, \ldots, i_{01}, \ldots, i'_k\right\rangle$ implies that the test is also verified for i'_k along $\left\langle i'_0, \ldots, i_0, i_1, \ldots, i'_k\right\rangle$.

3. As a last case, let us assume that the i'_k chosen is i_1 (and i'_{k-1} is i_0) but i_0 and i_1 do not appear consecutively in the subsequence $\left\langle i'_0, \ldots, i'_{k-1}\right\rangle$. In that case, the sufficiency test for i'_k in the original routine would have been:

$$r \in expected\left(i_1, m\right) \Rightarrow r \in def\left(i_0, m\right) \vee \exists i_x \in W' : r \in def\left(i_x, m\right) \tag{5.4.7}$$

   where $W' = \left\{i'_j | j = 0,\ldots,k-1\right\} \setminus \{i_0\}$. In the modified routine body, however, i_1 is no longer present and no similar condition is tested. However, the instruction following i_1, which we called i_2, is also reachable and, since sufficiency checks all reachable instructions, it is true that the following condition will also need to be verified:

$$r \in expected\left(i_2, m\right) \Rightarrow r \in def\left(i_{01}, m\right) \vee \exists i_x \in W' : r \in def\left(i_x, m\right) \tag{5.4.8}$$

   It is desirable to determine whether this last condition does imply the condition 5.4.7 above, or what additional condition needs to be verified as well. Let us consider some subcases:

(a) If $r \in expected\,(i_1, m)$ and also $r \in expected\,(i_2, m)$, then we are recursively in the case (2) above, considering i_2 as i'_k. We obtain that $r \in def\,(i_0, m) \vee r \in def\,(i_1, m) \vee r \in W'$, satisfying the right hand side of expression 5.4.7. Consequently, if $r \in expected\,(i_1, m)$ and $r \in expected\,(i_2, m)$, expression 5.4.8 implies expression 5.4.7.

(b) The only remaining case is $r \in expected\,(i_1, m)$ but $r \notin expected\,(i_2, m)$. The definition of $r \in expected\,(i_1, m)$ says that

$$r \in expected\,(i_1, m) \iff r \in use\,(i_1, m) \vee \left(r \notin \bigcup_{\forall m' \in M} def\,(i_1, m) \wedge r \in expected\,(i_2, m) \right)$$

and consequently, if $r \notin expected\,(i_2, m)$, it must be $r \in use\,(i_1, m)$. Summarising, the only case in which expression 5.4.7 could be not verified while expression 5.4.8 holds, therefore, is the case in which

$$r \in use\,(i_1, m) \wedge r \notin def\,(i_0, m) \wedge \nexists i_x \in W' : r \in def\,(i_x, m)$$

Now, let us assume that $r \in use\,(i_1, m)$, $r \notin def\,(i_0, m)$ and that there is no $m' \in M, m' \neq m$ such that $r \in def\,(i_0, m')$. Recalling that

$$r \in use\,(i_{01}, m) \iff r \in use\,(i_0, m) \vee \left(r \in use\,(i_1, m) \wedge r \notin \bigcup_{\forall m' \in M} def\,(i_0, m') \right)$$

we would also obtain that $r \in use\,(i_{01}, m)$. However, $r \in use\,(i_{01}, m)$ also implies $r \in expected\,(i_0, m)$, and, by sufficiency, we would have once again that $\exists i_x \in W' : r \in def\,(i_x, m)$. Therefore, the only case in which expression 5.4.7 is false, while expression 5.4.8 is true, is:

$$r \in use\,(i_1, m) \wedge r \notin def\,(i_0, m) \wedge \exists m' \in M, m' \neq m : r \in def\,(i_0, m')$$

Finally, it is now possible to say that sufficiency is verified on the original routine body if it is verified on the routine containing i_{01} instead of i_0,i_1 and, additionally:

$$\forall m_1, m_2 \in M, m_1 \neq m_2 : def\,(i_0, m_1) \cap use\,(i_1, m_2) = \varnothing$$

This last condition is, once again, the very same condition 5.4.6 previously found in the case of validity. Verifying such condition, in addition to validity and sufficiency of the modified routine, it is therefore also possible to determine that the original routine is both valid and sufficient. Since condition 5.4.6 does not depend on the value of *expected*, the test can be performed during the combination of i_0 and i_1.

5.4.6.3 Consistency of *def* and *use*

The definition of consistency for *def* and *use* on i_0 and i_1, according to Condition 5.2.4, is:

$$\forall m_1, m_2 \in M, m_1 \neq m_2, \forall i \in \{i_0, i_1\} : \begin{cases} use(i, m_1) \cap use(i, m_2) = \varnothing \\ def(i, m_1) \cap def(i, m_2) = \varnothing \end{cases}$$

The verification of these conditions is local to each instruction, and does not rely on the values of *expected*. Expanding the definitions of $use(i_{01}, m_1)$ and $def(i_{01}, m_1)$, it is straightforward to verify that:

$$\forall m_1, m_2 \in M, m_1 \neq m_2 : use(i_{01}, m_1) \cap use(i_{01}, m_2) = \varnothing \Rightarrow \\ use(i_0, m_1) \cap use(i_0, m_2) = \varnothing$$

$$\forall m_1, m_2 \in M, m_1 \neq m_2 : def(i_{01}, m_1) \cap def(i_{01}, m_2) = \varnothing \Rightarrow \\ def(i_1, m_1) \cap def(i_1, m_2) = \varnothing$$

In order to verify the consistency of def and use on i_0 and i_1, therefore, it is enough to verify the consistency of the definitions of def and use on i_{01} and, additionally, to check that:

$$\forall m_1, m_2 \in M, m_1 \neq m_2, \begin{cases} use(i_1, m_1) \cap use(i_1, m_2) = \varnothing \\ def(i_0, m_1) \cap def(i_0, m_2) = \varnothing \end{cases} \tag{5.4.9}$$

In this case as well, the test does not depend on the value of *expected*, and can be easily performed during the combination of i_0 and i_1.

5.4.7 Delay slot elimination: transformed functions

It is now possible to describe formally the simple transformations that can be applied locally to each delayed instruction in order to obtain an equivalent representation of the program that does not contain delayed instructions.

5.4.7.1 Unconditional branch, delayed

If the instruction that is being considered is an unconditional branch, and a delay slot is used, the instruction that immediately follows the branch is actually executed before the value of the program counter is changed, as a consequence of the delayed branch execution. The effect is therefore equivalent to a sequence made up of an instruction that has no effect on the control flow, plus a an instruction that behaves as the original instruction in the delay slot, but whose following instruction, in the control flow, is the target of the original branch.

Case 1.

$$\forall i \in I, \forall m \in M : type(i) = uncond \wedge delay(i) = delayed \Rightarrow \begin{cases} use'(i,m) = use(i,m) \\ use'(succ(i),m) = use(succ(i),m) \\ def'(i,m) = def(i,m) \\ def'(succ(i),m) = def(succ(i),m) \\ follow'_0(i) = succ(i) \\ follow'_0(succ(i)) = null \\ follow'_1(i) = null \\ follow'_1(succ(i)) = follow_1(i) \end{cases}$$

5.4.7.2 Unconditional branch, annulled delay

In the case in which an annulled delay slot is used for an unconditional branch, the instruction in the delay slot is completely ignored, being equivalent to a 'nop' instruction. However, the branch is still delayed, and the value of the program counter changes only after the instruction in the delay slot has been fetched and ignored. Consequently, this is the set of definitions that describe this case.

Case 2.

$$\forall i \in I, \forall m \in M : type(i) = uncond \wedge delay(i) = annul \Rightarrow \begin{cases} use'(i,m) = use(i,m) \\ use'(succ(i),m) = \varnothing \\ def'(i,m) = def(i,m) \\ def'(succ(i),m) = \varnothing \\ follow'_0(i) = succ(i) \\ follow'_0(succ(i)) = null \\ follow'_1(i) = null \\ follow'_1(succ(i)) = follow_1(i) \end{cases}$$

5.4.7.3 Conditional branch, delayed

When a delay slot is used with a conditional branch, the instruction in the delay slot is executed in any case before the possible change in the value of the program counter. The only effect on the execution flow is that the branch target branch appears to be a follower of the instruction in the delay slot rather than of the branch itself. Consequently, the following definitions represent what happens in the case of a delayed conditional branch.

Case 3.

$$\forall i \in I, \forall m \in M : type(i) = cond \wedge delay(i) = delayed \Rightarrow \begin{cases} use'(i,m) = use(i,m) \\ use'(succ(i),m) = use(succ(i),m) \\ def'(i,m) = def(i,m) \\ def'(succ(i),m) = def(succ(i),m) \\ follow'_0(i) = succ(i) \\ follow'_0(succ(i)) = follow_0(succ(i)) \\ follow'_1(i) = null \\ follow'_1(succ(i)) = follow_1(i) \end{cases}$$

5.4.7.4 Conditional branch, annulled delay

The situation gets slightly more complicated when annulled delay slots are used in conjunction with conditional branches. In this case, the instruction in the delay slot is executed only if the branch is taken. Consequently, the branch target will appear to be a follower of the instruction in the delay slot, but the instruction after the delay slot will appear to be a follower of the branch. In detail:

Case 4.

$$\forall i \in I, \forall m \in M : type(i) = cond \wedge delay(i) = annul \Rightarrow \begin{cases} use'(i,m) = use(i,m) \\ use'(succ(i),m) = use(succ(i),m) \\ def'(i,m) = def(i,m) \\ def'(succ(i),m) = def(succ(i),m) \\ follow'_0(i) = follow_0(i) \\ follow'_0(succ(i)) = null \\ follow'_1(i) = follow_0(succ(i)) \\ follow'_1(succ(i)) = follow_1(i) \end{cases}$$

5.4.7.5 Call instruction, delayed

In the context of delay slots elimination, call instructions need to be treated in a rather particular way. As discussed in Section 5.2.8, the definitions of def and use for a call instruction summarize the effect of the call to subroutine as a whole. The use of delay slots however, as seen in Section 5.4.4, requires some additional care. During execution, a delayed call instruction will appear initially not to have any visible effect as the control flow steps through the call instruction to reach the instruction in the delay slot. Right after the instruction in the delay slot has been executed, however, the new value of the program counter will be loaded, and the subroutine will be called. As a net result, when the program counter returns to the instruction that follows the delay slot, the combined effects of the instruction in the delay slot and the effect of the entire call will appear

to have taken place simultaneously. The effect of the delay slot can be consequently described adjusting accordingly *def* and *use*, combining the instruction in the delay slot with the pseudo-instruction corresponding to the call instruction, as follows:

Case 5.

$$\forall i \in I, \forall m \in M : type(i) = call \wedge delay(i) = delayed \Rightarrow \begin{cases} use'(i,m) = \varnothing \\ use'(succ(i),m) = use(\langle succ(i), i\rangle, m) \\ def'(i,m) = \varnothing \\ def'(succ(i),m) = def(\langle succ(i), i\rangle, m) \\ follow'_0(i) = follow_0(i) \\ follow'_0(succ(i)) = follow_0(succ(i)) \\ follow'_1(i) = null \\ follow'_1(succ(i)) = null \end{cases}$$

This definition relies on the assumption that the call instruction, from the point of view of the microprocessor, does not use or define, in itself, any registers. If, for whatever reason, the call instruction on a certain architecture relies or has side effects on certain registers, it is sufficient to replace the empty sets above with the appropriate sets of registers.

5.4.7.6 Other cases

As required by Condition 5.4.5, instructions having values for *type* and *delay* equal to $(call, annul)$, $(simple, delayed)$ or $(simple, annul)$ cannot appear in I. The only other remaining cases involve either simple instructions that are not in delay slots, or branches and calls that are not delayed. The two possibilities are represented below:

Case 6.

$$\forall m \in M, \forall i \in I : \left(\left(i = i_{first} \vee \left(i \neq i_{first} \wedge delay\left(succ^{-1}(i)\right) = nodelay\right)\right) \wedge \right.$$
$$\left. \wedge type(i) = simple \wedge delay(i) = nodelay\right) \Rightarrow \begin{cases} use'(i,m) = use(i,m) \\ def'(i,m) = def(i,m) \\ follow'_0(i) = follow_0(i) \\ follow'_1(i) = follow_1(i) \end{cases}$$

Case 7.

$$\forall i \in I, \forall m \in M : (delay(i) = nodelay \wedge$$
$$\wedge type(i) \in \{cond, uncond, call\}) \Rightarrow \begin{cases} use'(i,m) = use(i,m) \\ def'(i,m) = def(i,m) \\ follow'_0(i) = follow_0(i) \\ follow'_1(i) = follow_1(i) \end{cases}$$

Finally it is necessary to define use' on the value $exit$: $\forall m \in M : use'(exit, m) = use(exit, m)$, and def' on the value $enter$: $\forall m \in M : def'(enter, m) = def(enter, m)$. That completes the definition of the auxiliary functions def', use', $follow'_0$ and $follow'_1$, that can then be used in the customised liveness analysis in order to determine the modes of the various registers. From an algorithmic point of view, the construction of the auxiliary functions can be performed in a single pass following the algorithm shown below.

5.4.8 The delay slot elimination algorithm

Since all the transformations can be performed locally, the algorithm can be used to build the definitions for def', use', $follow'_0$ and $follow'_1$ in a single linear pass over the set I. According to Section 5.4.6, some checks are also performed whenever it is necessary to combine multiple instructions, in particular conditions 5.4.9 and 5.4.6. To simplify the description, it will be assumed that the representations of the functions def, use, $follow_0$ and $follow_1$ are simply overwritten to reflect the new definitions def', use', $follow'_0$ and $follow'_1$. This is the structure of the algorithm:

Algorithm 3 – Delay slot elimination algorithm

procedure *delaySlotElimination*()

Uses: Map use defined on $I_e \times M$
Uses: Map def defined on $I_p \times M$
Uses: Maps $follow_0$, $follow_1$ defined on I_p
Uses: Maps $type$ and $delay$ defined on I

{Check Condition 5.4.4}
1: **if** $type\left[succ^{-1}(exit)\right] \neq nodelay$ **then**
2: **error**
3: **end if**
{Other conditions on delay slots}
4: **for all** $i \in I$ **do**
5: **if** $follow_1[i] \neq i_{first}$ **then** {Check Condition 5.4.2}
6: **if** $type\left[succ^{-1}(follow_1[i])\right] \neq nodelay$ **then**
7: **error**
8: **end if**
9: **end if**
10: **if** $delay[i] \neq nodelay$ **then** {Check Condition 5.4.3}
11: **if** $type[succ(i)] \neq simple$ **then**
12: **error**
13: **end if**
14: **end if**
15: **if** $type[i] = simple$ **then** {Check first part of Condition 5.4.5}
16: **if** $delay[i] \neq nodelay$ **then**
17: **error**

```
18:       end if
19:    end if
20:    if type[i] = call then {Check second part of Condition 5.4.5}
21:       if delay[i] = annul then
22:          error
23:       end if
24:    end if
25: end for
    {follow_0 is generated and follow_1 checked according to Section 5.4.1}
26: for all i ∈ I do
27:    if type[i] = simple ∨ type[i] = call then
28:       if follow_1[i] ≠ null then
29:          error
30:       end if
31:       follow_0[i] ← succ(i)
32:    else if type[i] = cond then
33:       if follow_1[i] ≠ null then
34:          error
35:       end if
36:       follow_0[i] ← succ(i)
37:    else if type[i] = uncond then
38:       if follow_1[i] ≠ null then
39:          error
40:       end if
41:       follow_0[i] ← null
42:    end if
43: end for
    {All preconditions are verified. It is now possible to eliminate the delay slots}
44: i ← i_first
45: while i ≠ exit do
46:    if type[i] = uncond ∧ delay[i] = delayed then {Case 1}
47:       follow_0[i] ← succ(i)
48:       follow_0[succ(i)] ← null
49:       follow_1[succ(i)] ← follow_1[i]
50:       follow_1[i] ← null
          {Skip delay slot}
51:       i ← succ(i)
52:    else if type[i] = uncond ∧ delay[i] = annul then {Case 2}
53:       for all m ∈ M do
54:          def[succ(i), m] ← ∅
55:          use[succ(i), m] ← ∅
56:       end for
```

57: $follow_0[i] \leftarrow succ(i)$
58: $follow_0[succ(i)] \leftarrow null$
59: $follow_1[succ(i)] \leftarrow follow_1[i]$
60: $follow_1[i] \leftarrow null$
{Skip delay slot}
61: $i \leftarrow succ(i)$
62: **else if** $type[i] = cond \wedge delay[i] = delayed$ **then** {Case 3}
63: $follow_1[succ(i)] \leftarrow follow_1[i]$
64: $follow_1[i] \leftarrow null$
{Skip delay slot}
65: $i \leftarrow succ(i)$
66: **else if** $type[i] = cond \wedge delay[i] = annul$ **then** {Case 4}
67: $follow_1[succ(i)] \leftarrow follow_1[i]$
68: $follow_1[i] \leftarrow follow_0[succ(i)]$
69: $follow_0[succ(i)] \leftarrow null$
{Skip delay slot}
70: $i \leftarrow succ(i)$
71: **else if** $type[i] = call \wedge delay[i] = annul$ **then** {Case 5}
72: **for all** $m_1 \in M$ **do**
73: **for all** $m_2 \in M, m_1 \neq m_2$ **do**
74: **if** $def[i,m_1] \cap def[i,m_2] \neq \varnothing$ **then**
75: **error**
76: **end if**
77: **if** $use[succ(i),m_1] \cap use[succ(i),m_2] \neq \varnothing$ **then**
78: **error**
79: **end if**
80: **if** $def[i,m_1] \cap use[succ(i),m_2] \neq \varnothing$ **then**
81: **error**
82: **end if**
83: **end for**
84: **end for**
{The sequence $\langle succ(i), i\rangle$ is combined, i_0 is $succ(i)$ and i_1 is i}
85: $udef0 \leftarrow \varnothing$
86: $udef1 \leftarrow \varnothing$
87: **for all** $m \in M$ **do**
88: $udef0 \leftarrow udef0 \cup (def[succ(i),m])$
89: $udef1 \leftarrow udef1 \cup (def[i,m])$
90: **end for**
91: **for all** $m \in M$ **do**
92: $def[succ(i),m] \leftarrow def[i,m] \cup (def[succ(i),m] \setminus udef1)$
93: $use[succ(i),m] \leftarrow use[succ(i),m] \cup (use[i,m] \setminus udef0)$
94: $def[i,m] \leftarrow \varnothing$

95: $use[i,m] \leftarrow \varnothing$
96: **end for**
{Skip delay slot}
97: $i \leftarrow succ(i)$
98: **end if**
{No delay ⇒ nothing to do}
99: $i \leftarrow succ(i)$
100: **end while**

The complexity involved is linear, both in space and in time, with respect to $|I|$ (considering the sets R and M fixed). The algorithm can be optimised, during implementation, to group together similar instructions and loops, which are separate in this description in order to show more clearly the various conditions and operations. Once the algorithm is complete, the information encoded in the function *type* and the branch targets described by $follow_1$ are encoded in the new $follow'_0$ and $follow'_1$, which can be used for the normal mode analysis algorithm.

When eating an elephant, take one bite at a time.
*— **Gen. C. Abrams**, 1914-1974*

Chapter 6

Pointer Discovery in the Stack

As we have seen in the previous chapters, tracking pointer information, or more generally mode information, in the registers of the microprocessor for every machine instruction presents a number of challenges. Similarly, discovering all the pointers that may be present in the stack presents a number of difficulties, due to the fairly complex arrangement of the layouts that the stack frames may have. For instance, on each stack frame there might be local variables, temporary values, saved registers and so on. This chapter will present the various issues related to determining the location of all the pointers present on the stack and some techniques that can be of use. Chapter 9 will describe the specific implementation details of the current prototype.

6.1 Stack components

In the typical implementations of many high-level languages, the stack for a single thread is treated as a sequence of activation records, also called frames, corresponding to the various nested routine invocations that are active at any given moment, from the main routine to the one most recently called. Each activation record is used to store information related to one invocation of a routine. While some of the information mainly depends on the routine's local data (parameters, automatic variables and so on), some other information is more closely related to the specific implementation scheme (registers save area and dynamic links, for instance). The exact structure of each activation frame is decided by the compiler according to the the optimisations applied and its internal rules, including the choice of whether variables, parameters and other data are allocated on the stack or in the registers.

Regardless of the specific optimisations performed, however, the overall frame layout is constrained by the Application Binary Interface (ABI) for the specific microprocessor architecture in use. Such specifications are designed to simplify interoperability among different programming languages and compilers, specifying, among the rest, calling conventions and the location and format of the implementation-related data of which the system should be aware, for instance during

interrupts. The standard mechanisms established by the ABI offer a certain degree of separation between caller and callee, making it easier to determine the set of live pointers in each frame, with a reasonable degree of accuracy, considering each routine in isolation. An inter-procedural analysis could allow, in certain cases, a higher degree of precision, but it would also require a great deal of additional work. The following discussion will focus on intra-procedural techniques.

As previously mentioned, several different components may be present in every stack frame. Those parts will now be listed, and the ways to discover pointers for each of them discussed. Techniques used more specifically for the SPARC in the current prototype will be discussed later, in Chapter 9.

6.2 Return addresses

Each time a subroutine is called, the routine that performs the call is temporarily suspended. After the callee has completed its job, control must returns to the caller, which implies that the value of the program counter at which the execution of the caller was suspended has to be preserved. That value, the return address, is used to determine where to return control after the end of the called subroutine.

The most obvious location to save the current value of the program counter is the stack frame of the called routine, so that it can be retrieved from there when returning. Depending on the microprocessor architecture and on the ABI, the last return address, or the last few, can alternatively be stored in some of the available registers, for greater efficiency [Mot93, Spa94]. If the call chain is deep enough, however, the registers dedicated to store the return addresses will sooner or later have to be reused. For this reason, a dedicated portion of each activation record is usually also reserved to store the values of the return addresses. If the registers that contain return values need to be reused, their content is copied in the corresponding locations in the reserved part of the stack frame.

Notably, those routines that do not in turn call other routines, also called “leaf” routines, are often treated in a special way. In the case in which, for instance, a reserved register is used to store the last return address, a leaf routine will never need to save the content of that register somewhere else in order to make further calls. As a consequence, there is no need to reserve a slot on the stack frame for that purpose. It is not unusual for ABIs to prescribe a different standard format for the layout of the frames of leaf routines, in order to make a more efficient use of available registers.

The various locations on the stack used to store return addresses, and the reserved registers, can contain valid pointers to executable code, but never valid pointers to heap objects. Their values, therefore, only need to be adjusted if the executable code they refer to is dynamically relocated, but no adjustments are necessary in order to support heap manipulations.

It should also be noted that, if a call instruction is immediately followed by a return instruction, a tail call optimisation can be performed by the compiler. In that case, both the call instruction and the return instruction that follows are replaced with a plain branch, and the parameter passing sequence is adjusted accordingly. While a tail recursion is essentially a branch within the same

compiled routine, a tail call to a different routine ("sibling call") may look slightly different, depending on the architecture in use, from a conventional call. For instance, parts of the current stack frames could be overwritten to store the arguments for the new routine. While sibling call optimisations do not alter the possibility to analyse the various routines in isolation, the modified call sequence should be treated with special attention in order to reconstruct correctly the sequence of activation records. Other forms of optimisations (inlining, for instance) may also cause the call sequence of compiled routines not to reflect precisely the structure of the original program, but that is inessential as far as pointer discovery is concerned. We are here interested in the structure of the compiled code rather than in the functions and procedures of the original source program.

6.3 Dynamic chain

As we have seen, saving the program counter is a crucial step in calling a subroutine, and it is important to have space allocated for that information on the stack. However, in order to know where the return addresses are saved, it is also essential to know at all times how to retrieve the location of the stack frames used by the various routines. Depending on the ABI of the microprocessor in use, there might be different ways to determine the address of the current activation record, but for efficiency reasons that address is customarily stored, during execution, in a dedicated register known as frame pointer. This register is saved on the stack, or in further registers, as control moves across routines, so that it constantly refers to the activation record of the current routine. The pattern followed is the same previously described for the program counter and the return addresses. In the same way in which the return addresses point within the code of routines, describing the chain of successive invocations, the saved values of the frame pointer (the "dynamic links") define a chain of activation records, the "dynamic chain".

Sometimes, if no semi-dynamic variables are used (see later), it is possible to determine the size of each stack frame statically. In that case, the value of the frame pointer can be calculated using the current value of the stack pointer and fixed offsets. Consequently, the stack pointer can replace the frame pointer as a base register for the accesses to the stack frame contents, and the register that is normally reserved to store the frame pointer can be reused for other purposes. As a side effect, there is no need to reserve space on the stack to store the values of the frame pointer.

Some architectures, including the SPARC, offer hardware facilities to save and restore automatically the value of the frame pointer while creating and deleting activation records. As in the case of the return address, the value of the frame pointer need not be saved, in certain cases, when leaf routines are used. If no further local information is required on the stack, leaf routines can be implemented on some microprocessors without creating an activation record at all. It is necessary to pay special attention to this possibility in order to attribute correctly each activation record to the matching compiled routine while traversing the dynamic chain.

In order to discover all of the pointers in all the active stack frames at any instruction, the full dynamic chain must be known whenever the microprocessor is preemptively interrupted. Surprisingly, however, on certain microprocessor architectures it is extremely difficult to discover the exact state of the dynamic chain while executing certain instruction sequences in perfectly

ordinary programs. On the SPARC, for instance, part of the necessary information is kept in some registers, but inspecting registers and stack is not enough to determine which ones are in use at any given moment. More details on this aspect are available in Section 9.7.1.

While the dynamic links are valid pointers, they are also known to be pointers to the stack. Their values, consequently, only have to be adjusted if, for some reason, the stack needs to be dynamically relocated.

6.4 Static chain

In the languages that support lexical nesting, it is necessary to maintain information about the most recently activated stack frames of those routines that lexically enclose the currently active routine. In order to keep track of those frames, a common solution is to arrange them in a chain, known as the static chain, in which each stack frame contains a reference, the static link, to the most recent stack frame of the enclosing routine. The chain, therefore, goes from the frame used by the innermost routine to the one used by the outermost routine, following the static links.

The static chain is used to retrieve the automatic variables of the enclosing routines. For instance, in order to retrieve the variables that are defined in the routine immediately enclosing the current one, the first static link is used in order to find the correct activation record. If the frame of the routine enclosing the one enclosing the current one (two levels of nesting) is required, two steps of the static chain are followed, and so on.

While the static chain is easily implemented by adding a slot to each activation frame, it is usually more efficient to use a single table, the display vector, to store the references to those frames, one vector position per nesting level. Since the number of static links is not influenced by recursion, but only by the static nesting structure of the program, a limited number of positions is sufficient [ASU86]. Static links, which are legitimate pointers, could therefore be present on the stack (or in the display vector), assuming values corresponding to stack addresses. Considerations similar to those made for dynamic links apply, and we can conclude that heap manipulations do not affect the elements of the static chain.

6.5 Arguments

Arguments can be passed to the called subroutine in a variety of ways. However, once again to guarantee interoperability, a single passing convention is established by the ABI. Broadly speaking, on common microprocessors arguments are either passed entirely on the stack, or some in the registers and the remainder on the stack. The first approach is preferred by those microprocessors that have a very limited number of registers, while the second is favoured by those with a more extended register set. In any case, part of the tuple of arguments might end up on the stack, and some pointers might be contained there. It is therefore necessary to discover those pointers in order to perform the necessary adjustments when required. The rest of this section focuses on arguments passed in the stack.

The exact calling conventions and the argument passing mechanism are extremely dependent on the microprocessor's features. In general, outgoing arguments are set up before the before the subroutine is called, and before the new activation record is created. Consequently, the arguments normally reside in the caller's frame.

From a conceptual point of view, each argument has a lifetime both in the caller and in the callee, and as such some mode information must be available when the microprocessor is interrupted in either routine. For the caller, the lifetime of a stack slot used as an outgoing argument extends from the moment in which the argument is stored to the moment in which the call is made (assuming call by value). The callee sees the same stack slot as an incoming argument, whose lifetime extends from the routine prologue down to the last use of that slot in the routine body. As a consequence, descriptors for the mode of such stack slots will have to appear in both routines.

A useful distinction could be made between those microprocessors (such as the x86 family) which have PUSH/POP-style operations, and those that rely instead on addressing formats with displacement to access the various stack slots. If PUSH and POP are used, the general trend is to use those operations to push the necessary arguments onto the stack, one by one. In that case, the value of the stack pointer keeps moving up and down following the accumulation of arguments and their release upon return. If, on the other hand, PUSH and POP are not available, the necessary arguments will be stored in the current fixed-size frame, and the value of the stack pointer does not move. The adjustments of the stack pointer need to be taken into account if a request for a service routine, and consequently for pointer discovery, is made while the parameters are being pushed. In particular, some compilers perform rather tricky optimisations in order to reduce the number of POP operations. For instance, calling two routines with one argument each would normally be represented as: PUSH(arg1), CALL(x), POP(1), PUSH(arg2), CALL(y), POP(1). But a optimising compiler could change the sequence to: PUSH(arg1), CALL(x), PUSH(arg2), CALL(y), POP(2). The exact offset by which the stack pointer has moved with respect to the value assumed at the beginning of the routine body, should be determined (statically) for every machine instruction. That information can then be used at runtime to reconstruct, from the current value of the stack pointer, its initial value, which can therefore be used as a reference point within the frame.

Regardless of the optimisation applied, and whether the stack pointer moves during execution or not, it is always possible to keep track of the various locations used to store outgoing arguments calculating their offset with respect to the initial value of the stack pointer.[1] Keeping track of the offsets, it is still possible to perform a liveness analysis similar to the one used for registers, and generate a data structure that can be used to associate each value of the program counter to the current mode of each frame slot used as an outgoing argument. More details are available in the discussion on automatic variables, in Section 6.7.

When the same slots are treated as incoming arguments, as seen by the callee, they can be considered in a manner entirely similar to automatic variables, except that they are already defined

[1] A recent contribution made by Josef Zlomek of SUSE Labs and Daniel Berlin of IBM Research, for instance, improves the ability of GCC to track variables in the cases in which the stack pointer changes during the execution of the routine body, as described above. Such information can then be used by the GDB debugger to keep track of variables in optimised code.

at the beginning of the routine body. The precise way in which incoming arguments are accessed by the callee depends on the specific details of the ABI, but generally they can be retrieved using fixed offsets from either the stack pointer or the current frame pointer. Particular attention must be paid to the fact that the incoming arguments are alive during the execution of the routine prologue. Since incoming arguments contained in stack slots are usually not touched by the prologue, their mode can be obtained by checking the tables or other structures that describe the use of those slots in the caller.

A particular case is represented by variadic argument lists, such as in the C printf function. While the caller knows which arguments are stored on the stack at each call site, the callee has no way of knowing statically which arguments, and of which types, will be used at runtime. The callee will access the variadic portion of its argument list as a generic list of values, or bytes, depending on criteria that bypass the usual type-system of the language. In particular, the compiler has no knowledge about those values while compiling the callee.

Leaving aside the usual criticisms of variadic argument lists, the feature poses a problem for this analysis in that it is not possible to determine statically neither the number of arguments nor their mode. It is however possible, at least in principle, to act conservatively on the variadic portion of the argument list knowing that the arguments will be accessed from a certain base register (the stack pointer, the frame pointer or something else) using an index, in a manner similar to an array. Full precision, however, cannot be obtained in general.

Variadic argument lists are a rare (and questionable) feature that can be easily replaced using different programming styles, and that are quite rare outside C/C++ [2]. Even though C99 offers a way to write variadic functions and macros, the original definition of C did not even offer a portable way to create user-defined variadic functions [KR88]. In practice, the main application of variadic argument lists is supporting traditional C library functions like `printf()` and `scanf()`. For such functions, in which the first argument is used as a format string, it is not far-fetched to imagine an implementation in which the first argument is parsed whenever an interrupt is received, in order to discover dynamically where pointers really are in the variadic argument block. That would allow once again to obtain full precision, even for those library functions. For comparison, it is worth pointing out that some compilers (like GCC) do actually inspect the format string during compilation, if statically available, to discover inconsistencies between the format string itself and the types of the remaining arguments.

It is also important to mention that, although arguments are usually either pointers or values of primitive types, it is actually possible, in many languages, to pass whole records as arguments. In that case, issues similar to those discussed in Section 2.4 must be taken into account, with particular reference to variant and packed records. Moreover, if the structure is very large, a

[2] Variadic argument lists (varargs) are a planned feature of the upcoming Java 2 Platform, Release 1.5 [BB]. They are however treated as `Object[]` arrays, automatically created by the compiler, and as such the underlying implementation will use a single argument containing a pointer to the array object. Consequently, there are no problems similar to C/C++ varargs, in which each element of the variadic list has to be a single and independent argument because of compatibility with other functions. The feature remains controversial in Java, with some advocating Compact Object Array Literals (COAL) instead, as in `foo(1,2,{8,9.2,"aa"},3,{"bb",7});` where the elements in braces are automatically transformed in `Object[]` arrays. Using the latter notation, the number of arguments is actually fixed. Since Java 1.5 is designed to be bytecode-compatible with Java 1.4, the new changes and features are in any case only visible at the source code level.

copy operation could be used by the compiler to set up the large argument, using either block-move instructions (if available), a system routine, or a simple loop. In some cases, an interrupt could find the argument only half-copied, and it could be quite difficult to determine how much of the structure has been copied without inspecting loop counters and similar implementation details. Determining exactly which locations contain live pointers, therefore, may prove to be rather difficult.

There are two simple approaches that can be used in those cases. The simpler course of action, of course, is simply treating the copy operation as a critical section. If an interrupt is received halfway through the copy, execution is simply resumed until the end of the loop. On the other hand, if a quicker response is desired, it is also possible to work conservatively, while maintaining the ability to move memory about the heap. The layout of the destination record, which will be used as an argument, is known statically. Since no location is shared between pointers and scalars, a slot which is dedicated to a pointer can either really contain a pointer, or be unused, but it will never contain valid scalar data. Consequently, even if the record is only partially copied and some random data are mistakenly identified as valid pointers, they can be freely altered as a result of heap memory movements. Such an approach is conservative, since certain memory blocks could mistakenly appear to be in use and not be reclaimed, but it is still possible to move memory and adjust pointers freely.

It should also be noted that in certain cases the conventions defined by the ABI for parameter passing, when records are used as parameters, might require a copy of the original record to be stored on the stack, but then a pointer to that copy is passed to the callee, for instance in a register, as a reference to the argument. If such a pointer is in turn stored on the stack, than it should be taken into the account that one further stack location contains a valid pointer, although its value refers to a stack location rather than a memory block in the heap.

Finally, there might be the case in which an array is passed by value as an argument, which requires the whole array to be copied. The treatment is the same as for records. In the particular case of array arguments passed by value with parametric length, as allowable for instance in many Pascal extensions, the length is generally available at runtime as a parameter to the callee, and it is again possible to determine the layout of the argument area and the location of pointers at runtime. More details on this aspect are contained in Section 6.7.6. In the case of C/C++, the real length of arrays passed as parameters is not available, but arrays are treated as pointers and whole arrays cannot be passed by value as arguments. Similarly, Java arrays are actually complete objects, and Java objects are handled by implicit reference rather than by value.

6.6 Return value

Considerations entirely similar to those made for arguments apply to the return value, which is also shared between the callee and the caller. For efficiency reasons, it is common practice to use some conventional registers to store the return value. However, in the case of a complete structure (record or array) used as a return value, a different convention may apply. The resulting structure is in those cases usually stored in an area reserved somewhere on the stack, and a pointer is used

to communicate the location of the result between caller and callee. As in the case of arguments, the possible presence of a structure on the stack, used to store the return value, must be taken into account while statically creating the descriptors that will be used to discover the pointers at runtime.

In an object-oriented language that returns an object to the caller, the most common scenario is the implementation returning a pointer to the resulting object, allocated in the heap. In the (unlikely) case of an implementation which actually allocates such object on the stack, there would be a potential problem. Since the full type of the object returned is not known statically, determining the number and the position of the pointers is somewhat more difficult. However, since it must be possible to identify the real type of the object at runtime, there must also be a way to obtain, from the object, a pointer to the class descriptor. Using such information it is straightforward to determine the full object layout at runtime, and consequently the full set of pointers.

6.7 Automatic variables

Every activation record is also used to store automatic variables[3] for each routine invocation. While the compiler can use some of the registers to store the automatic variables which are more frequently used, the activation record is the obvious container for the remaining ones. In particular, each activation record could contain semi-static variables (fixed length) and semi-dynamic variables (such as parametric arrays). Additionally, certain automatic variables could be alive only for part of the routine, and multiple variables, with different modes, could share the same locations in the activation records. The various issues related to pointer discovery in the stack-allocated automatic variables will be now analysed.

6.7.1 Frame variants

While compiling a given routine, the compiler internally creates a map from the set of stack-allocated automatic variables to offsets in the activation records. Those offsets are then used to generate accesses to the variables in the corresponding compiled code. In order to save memory, it is common practice to reuse the available locations of an activation record, so that the same locations represent distinct variables that are never alive simultaneously. For example, consider the following C fragment.

```
if (t<10) {
 int a=t;
} else {
 int *x=&t;
```

[3] The term “automatic” is used here in the conventional sense of variables which are automatically created when a routine or block is entered, and discarded at the end. Note that the term “local”, although generally used to refer to automatic variables, could also refer to static local variables (in the C sense), which are not part of the activation record.

```
}
```

The fragment on the previous page is compiled by GCC, without applying optimisations, into the following SPARC code:

```
        ld [%fp+68], %o0
        cmp     %o0, 9
        bg      .LL3
        nop
        ld      [%fp+68], %o0  # extract value of t
        st      %o0, [%fp-12]  # store in automatic var a
        b       .LL4
        nop
.LL3:
        add     %fp, 68, %o0   # extract addr of t
        st      %o0, [%fp-12]  # store in automatic var x
.LL4:
```

The memory locations at the offset -12 in the frame is used for both the variables `a` and `x`, and in this case the mode of the same stack location is different depending on the position in the code.

In order to discover all the pointers in use at a certain instruction, at runtime, it will be necessary to have enough information available to map the instruction to the offsets which are reserved, at that stage, for pointers. Such information, which is available statically, can then be encoded in some data structure (a table, for instance) for later perusal. Of course, knowing that a stack location is reserved for a pointer does not strictly imply that a valid pointer is actually contained, or that it is alive, in that stack location at any given moment. For instance, in an array of pointers, not all elements might be in use at any given moment. A liveness analysis, where applicable, can be quite useful to refine the available information further, and more details will be discussed on this point later.

In principle, the necessary information can be obtained by inspecting the internal structures of the compiler. After the code generation pass the compiler knows both the offsets used by each automatic variable, and which assembler instructions resulted from the expansion of intermediate constructs that use certain variables. Consequently, it is also possible to create an association between the ranges of assembler instructions and the offsets which are used as pointers for each range. It is possible to think of this association as a series of “stack frame variants”, multiple layouts which differ in certain parts. Only one of those variants exists at each assembler instruction in the final compiled routine. The variants could be represented, for instance, by a tree in which the common part is completed by multiple alternatives for the variant parts, each of which in turn can be subdivided into its common and variant subparts, and so on. Such an organisation would also reflect the static nesting of lexical blocks within the source code. Other data structures describing the same information can also be used. At least in principle, the size of all the descriptors together could be quite large, being proportional to the number of assembler instructions times the number of frame slots used to store automatic variables. In practice, however, it is reasonable to expect that only a limited number of variants are used in every routine, due to practical considerations about programming styles.

Should table size become a concern, however, there is an alternative that can be used, at the expense of an increase of the frame record size. The crucial requirement is the ability to distinguish the slots which are used for pointers from those which are used for scalar data. That can be accomplished by modifying the offsets so that such sharing is never possible throughout the life of the routine. In that case, there would be just one possible layout, but it would be less economical in terms of space in the activation frame.

The idea of avoiding overlaps was used, for instance, by the JBE implementation mentioned in Appendix A. It should be noted that, if offsets are reorganised, it may be worth packing together all of the slots used for pointers, so that the specification of the layout is simply the size of the pointers area. There are also security implications in the idea, as pointed out by Hiroaki Etoh [Eto03], describing a research project in which "The novel features are [...] the reordering of local variables to place buffers after pointers to avoid the corruption of pointers that could be used to further corrupt arbitrary memory locations [...]". A related technique, used in the context of memory tagging rather than static handling, was used by Ganesan to reduce the tagging overhead [Gan94].

It was previously mentioned that only one of the variants may be in use at any one point in the code. That is the case, in concrete terms, if the final compiled code only uses a single mode for each stack slot at each point in the code, regardless of the dynamic path followed. In this sense, the requirement is similar to the "Gosling Property", as cited by Agesen, Detlefs, and Moss in their interesting papers on type-precision in Java [AD97, ADM98]. A pathological compiler could create code that does not respect that property, but it is reasonable to expect ordinary compilers to generate suitable code. In particular, GCC uses pseudo-registers to translate automatic variables, and in the final stages of the code generation those pseudo-registers are assigned to stack locations. Each stack slot is only assigned once to each live pseudo-register at one point of the code, therefore the above requirement is implicitly respected, at least for simple types. Additional sanity checks were however introduced in the experimental implementation, as described in Section 5.3. In the case of aggregate types, the property is still respected, except for variant records (known in C as unions). In such case, either variant records must be disallowed, or an offset adjustment must be introduced to make sure that there is never overlapping between pointers and scalars.

It might be useful to recall that this discussion assumes that the original optimised compiled code is left as much as possible unchanged and that the data structures used for the pointer discovery are created statically whenever possible, for a later dynamic inspection. Other solutions which imply an active tracking of pointers, as for instance memory tagging, are also possible but likely to be more expensive in terms of execution speed unless specialised hardware support is available. More details on the extraction of layout information are discussed later, in Section 6.7.7.

6.7.2 Liveness of variables or components with fixed offset

The automatic variables area of an activation frame, as previously discussed, is used to store data according to the low-level representation of the automatic variables originally present in the source code. For instance, simple data, like integers, floating-point numbers, and pointers, might

be present as the result of a straightforward translation of simple variables used in the source code. In principle, the same variable could potentially be allocated to different locations in two different portions of the code, if the compiler established that the variable is not alive somewhere in the middle and has reused the previous frame location. In practice, however, it is easier, while writing compilers, to maintain a fixed mapping from each automatic variable to a certain offset.

During the code generation, the compiler creates the final assembly code using, in accordance with the capabilities of the specific microprocessor, some sort of addressing mode in order to access the location associated with a certain automatic variable at that point in the code. The crucial aspect is that, regardless of the specific addressing mode in use and the possible intermediate calculations of pointers for the access, the compiler knows exactly the offset, in the activation record, at which the variable is located. In general it should be relatively straightforward to extract such information from the compiler, and generate local annotations similar to those used for the registers, with the purpose of describing which stack locations are written or read by each assembler instruction.

Using such local information, it is then possible to run a liveness algorithm similar to the one described in Chapter 5, using stack offsets rather than registers. Of course, as already discussed for registers, if the liveness information for the various stack slots is already present in the internal compiler structures at the level of detail of the single assembler instruction, a separate liveness analysis can be avoided. The resulting liveness map can then be inspected at runtime to discover not only which stack slots are reserved for pointers, as in the case of the frame variants described in Section 6.7.1, but also which slots actually contain live pointers. Such a simple analysis is perfectly suitable if automatic variables cannot be accessed by other routines, and it would be adequate, for instance, for the Java language. In such a case, each of the automatic variables would either have some primitive type or be an implicit reference to some object.[4]

The use of records on the stack, as long as arrays are not used, would also not cause any problems, since any access to a record component, arbitrarily nested, would still take place using a fixed offset. However, there are a number of cases in which local variables can be accessed using different mechanisms, which require some adjustments to the simple liveness analysis previously described. The most common cases are:

nested subroutines: if lexical nesting is available, the nested subroutines can freely access and overwrite the local variables of the enclosing routine.

[4] If inner classes are used, nested subroutines can actually access automatic variables of enclosing contexts if those variables are declared final (that is, they are constant after initialisation). From the Java Language Specification: “Any local variable, formal method parameter or exception handler parameter used but not declared in an inner class must be declared final, and must be definitely assigned before the body of the inner class” [GJSB00, page 141]. Without that restriction, inner classes would behave like closures. What really happens in Java is that the constant value is implicitly *copied* into a hidden area in the instance of the inner class. That allows (apparent) references to the enclosing context to be made, allowing the activation record of the enclosing routine to be discarded even though the instance of the inner class, with its local copy, can live on. That is a rather tricky case, since the exact time of the copy can only be known looking at the implementation, but it is reasonable to expect the copy operation to be made during the initialisation of the instance of the inner class. The call to such initialiser should therefore be annotated to indicate to the liveness algorithm that certain variables might be accessed (read) during the call. More details on this aspect are discussed in Section 6.7.3.

references: taking the address of a variable or field and passing it to a subroutine, the subroutine has no knowledge of which variable is actually being read or written.

arrays: since an array element can be accessed using an index which is not known statically, it is also not possible statically to know exactly which element is read or written.

These cases will now be discussed, with a description of the way in which the liveness information can be reconstructed, with varying degrees of accuracy depending on the specific approach chosen.

6.7.3 Nested subroutines

In certain circumstances, the automatic variables of a routine might be read or modified by one of the subroutines that are executed while the activation record that contains those variables is active. The most straightforward case is the use of a lexically nested subroutine, which is able to access such variables using the information supplied by the static chain. In this case, although the offset of the variable which is being accessed is statically known (except in the case of arrays, see later), different calls to the same subroutine might result in different read or write operations, depending on the control flow followed in each case.

In order to include this information in the liveness analysis, and determine therefore when each frame slot contains a live pointer, there are different possible approaches. A possible technique, for instance, might be using an intra-procedural analysis to gather some information about the possible uses of each automatic variable. Such an analysis would be partly simplified by the fact that the set of all the possible subroutines that are lexically nested in a given routine, and that can alter one of its variables, is statically known.

Performing an intra-procedural analysis, however, is in general rather complex. Since the compiler has information about the automatic variables that can be accessed by other routines (it uses that information to perform the frame slot allocation), an alternative would be to inspect the list of automatic variables in the internal compiler structures. If a certain automatic variable was allocated to a certain frame slot, and that variable can be accessed from other routines, it will be sufficient to assume, conservatively, that the call first reads and then updates that slot, and include the relevant annotations in our fine-grain liveness analysis. Doing so will force the slot to be considered as alive at the call site. If the compiler has more detailed information, and knows about slots that can or cannot be altered by individual calls to subroutines (for instance, some system calls might be known not to access local variables), such information can be used to obtain even more accurate liveness information. Once again, the liveness information obtained in this way is just a refinement of the information already available from the layouts described in Section 6.7.1; a slot reserved for pointers can be freely altered even if the value it contains is not a valid pointer at a particular time.

6.7.4 Reference passing

The case in which local variables can be accessed indirectly, through references, is somewhat similar, with the exception that references can be calculated and there is no certainty of the exact variable that will be modified in a certain case. For example, consider the following C fragment:

```
if (x<5)
 p=&a;
else
 p=&x;
doSomething(p);
```

In this example, it is not possible to know statically whether a pointer to `a` or `x` will be passed to `doSomething()`, and it will be necessary to consider that both of them could be accessed. The kind of reference described can be created explicitly by the programmer, in C for example, or generated automatically by the compiler. A typical example of automatic creation of references is the natural implementation of the "variable parameters" of Pascal and Modula, in which a reference to the argument variable is passed to the callee. Notably, a reference to a record can be used to access indirectly an inner component, therefore passing to a subroutine a reference to a record has the effect of exposing for inspection or modification all of the pointers contained in the record itself.

In the case of automatic variables that are accessed by subroutines via dereferencing, it is necessary to make conservative assumptions about the way in which some of the local variables can be used, or to inspect the information available in the compiler internals, if available, in order to obtain more precise information about the life of those frame slots. The compiler, in particular, makes conservative assumptions about the life or the variables while allocating the stack slots. For instance the compiler can discover that the address of a certain variable is taken at some point, either implicitly or explicitly, and it might conservatively assume that, in the following code, that variable can be altered by the called subroutines. There might also be an explicit indication (for instance a "volatile" qualifier)[5] that the variable can change across calls.[6] It should be noted that an automatic variable whose address is taken can be indirectly altered by the local routine as well, for instance if the user dereferences a pointer containing the address of that variable.

Those conservative assumptions on the life of the variables are used by the compiler in order to decide the allocation of the variables to the slots in the activation record. We might be interested

[5] There is a distinction in the way the compiler would behave if it saw that an address of a variable is taken, versus a "volatile" specifier. In the first case, the compiler will assume that the variable can be altered by any subroutine, and will make sure that a value cached in a register is flushed to memory before the call takes place. On the other hand, a "volatile" qualifier is used to specify that the variable can be accessed at any time, for instance by an interrupt routine, and therefore the value of the variable is always flushed to memory as soon as possible and it is read again from memory every time the variable is accessed.

[6] In C, a pointer which contains the address of an automatic variable could potentially be used to access a different automatic variable, adding a certain offset. Such an use would most likely confuse the compiler, and our analysis as well. It should be noted, however, that using pointers in this fashion is not portable, and not even legal C. Even if the implementation allows such usage, the implementation might reorder the slots used for the various automatic variables, or store some local variables in registers, making such an indirect use of pointers very dependent from the compiler's behaviour at best, and not a good programming practice in general.

in further refinement of that information, if additional information can be obtained by looking at the way in which the stack slots are used by the individual machine instructions.

For example, let us assume that the address of an automatic variable is copied into a register, and that the same register is dereferenced, later in the code, to perform read and write operations. If it is possible to determine statically that the register does not change, in the meantime, the accesses performed dereferencing the register really refer to the variable whose address was taken. Consequently, even if the compiler may conservatively consider the variable as alive during the whole routine, we can statically determine that the variable is only used by specific machine instructions. Performing this sort of analysis on the machine code, however, can be quite complicated, and it is not clear whether the effort would justify the increase in accuracy, with respect to the simpler information offered by the frame layouts.

6.7.5 Arrays

Issues similar to the ones described in the previous section are also found while dealing with stack-allocated arrays. When arrays are used, it is in general impossible to determine statically which elements will be read or written at any given moment during execution. As a consequence, it is also virtually impossible to determine the life of individual elements in an array without performing some sort of dynamic tracking. That is of particular concern to us since, if the array in question has locations reserved for pointers, it is not possible to tell whether those locations contain valid pointers or some random data. That could happen not just for arrays of pointers, but also for arrays of structures containing pointers, or arrays of arrays containing pointers, and so on.

In order to have more precise indications on the life of the individual array elements, a few considerations can be made. First of all, the life of the array as a whole is a limit to the life of the individual elements. To clarify, let us assume that the space for an automatic array containing some pointers is permanently allocated, by the compiler, at a certain offset in the activation frame for the entire body of the routine. We might be able to discover, analysing the code, that all of the accesses to that array are limited to a certain portion of the code. It is then trivial to conclude that, in the remaining parts of the code, the array as a whole is never alive, and that consequently none of the pointers it contains can be alive. Of course, it must still be considered that the array could be accessed, as discussed in the previous subsection, by lexically nested subroutines or indirectly through references, either to the array or to one of its elements or subcomponents.

Another possible approach, which can be combined with the previous one, could be to pre-initialise all of the locations reserved for pointers in the array to a conventional value, for example zero. If the conventional value is not a valid pointer, it is possible to assert that all of the pointers in the array which are found to be equal to that value are certainly not alive in that moment. The array layout, as explained in Section 6.7.7, can be obtained either from the compiler internals or from the debugging information. The possible positions of pointers in the array can therefore be easily determined. The main drawback of a dynamic pre-initialisation, of course, is the overhead required by the initialisation code. Additionally, it is also necessary to pay special care about possible interrupts received during the initialisation stage, which would find the array only partially

initialised. This approach is however quite natural for those programming languages that initialise all the variables to a conventional value in any case, as for instance does Java.

6.7.6 Semi-dynamic variables

A special case is the use of semi-dynamic[7] automatic variables, that is variables allocated on the stack whose size depends on runtime values, and is calculated during the allocation of the activation record. The term refers in particular to semi-dynamic automatic arrays, like unconstrained Ada arrays, or the open arrays of Turbo Pascal 7 and Free Pascal [Bor, Can04]. Semi-dynamic automatic arrays present a potential problem when determining the frame layout, since their size is not known statically.

While adding support for semi-dynamic automatic arrays involves some extra work, it is not difficult to imagine possible ways in which, at runtime, the real size of the array can be determined. First of all, the languages that support semi-dynamic arrays usually offer some facilities to retrieve the real lower and upper bounds of the array at runtime, and those values must be available as long as the array is in memory. For instance, Ada has the `'first` and `'last` attributes for arrays. The real values of the bounds are typically stored next to the array itself.

Second, the information necessary to reconstruct the real size of the semi-dynamic array can also be extracted by the debugging information. For example, the DWARF-2 "Debugging Information Format" manual states that "the debugging information must provide consumers a way to find the location of program variables, determine the bounds of dynamic arrays and strings [...]" [DWA93, Sec.2.4]. Extracting such information, it is then possible to know the real content of the array at runtime, and eventually discovering all the pointers.

6.7.7 Extracting layout information

As previously mentioned, there are two basic ways to extract the information required to reconstruct the frame layout(s) used by a compiled routine. The most direct, and complex, way is to inspect the compiler's internals, decoding its internal structures in order to find the frame slots reserved for pointers in the allocation record for every machine instruction. This solution might require some substantial work, especially if dealing with a pre-existing compiler, given the potential complexity of the data structures that must be explored.

[7] There is a degree of ambiguity in the terms "dynamic" and "semi-dynamic", when applied to arrays, and the terminology is not very consistent in literature. Sometimes the dynamic aspect is intended to be the ability of resizing the array after allocation. In this sense, "dynamic" arrays would be, for instance, the "flex arrays" of Algol68 while Java arrays, which cannot change size after allocation but have no statically fixed size, are occasionally called "semi-dynamic". Sometimes, even Ada arrays are called "dynamic". In this thesis, however, the terms "dynamic" and "semi-dynamic" refer to the storage class used for the data [GJ86]: dynamic arrays are arrays allocated dynamically, and not necessarily at the time of the creation of the allocation record, while semi-dynamic arrays are part of the activation record. Accordingly, we consider Java arrays to be "dynamic", since they are allocated independently from the creation of the activation record, they can survive the method that allocated them and are normally heap-allocated. To remove every possible ambiguity, the term "semi-dynamic automatic" is used as a synonymous for "stack-allocated, of size not statically known"

An alternative approach, that essentially extracts the same information, is the use of a standardized format for the extraction of such information. In particular, there are a number of well-established and well-documented debugging formats that are likely to offer exactly the kind of information that is required, and much more. Furthermore, it is highly likely that every compiler has some built-in ability to output some of its internal information in one of the standard debugging formats.

The more common debugging formats are the traditional STAB format, COFF, MIPS debug (Third Eye, part of ECOFF), DWARF (Debug With Arbitrary Record Format, used in conjunction with the ELF object file format) and SOM (HP's object file and debug format, unrelated to IBM's SOM ABI), just considering Unix-like environments.

One of the most popular format is probably DWARF, because of its flexibility and the wide diffusion of the ELF format. There are a few variations on the format: DWARF-1, the current DWARF-2, and the upcoming DWARF-3 [GS04]. An inspection of the DWARF-2 manual [DWA93] reveals the full extent of the information that can be obtained. In particular, the debugging format uses *location expressions* and *location lists* to describe where the various objects (variables and other data) are located. What follows is an extract from the manual:

> Location descriptions can be either of two forms:
>
> 1. *Location expressions* which are a language independent representation of addressing rules of arbitrary complexity built from a few basic building blocks, or operations. They are sufficient for describing the location of any object as long as its lifetime is either static or the same as the lexical block that owns it, and it does not move throughout its lifetime.
>
> 2. *Location lists* which are used to describe objects that have a limited lifetime or change their location throughout their lifetime. Location lists are more completely described below.
>
> [...] Each entry in a location list consists of:
>
> 1. A beginning address. This address is relative to the base address of the compilation unit referencing this location list. It marks the beginning of the address range over which the location is valid.
>
> 2. An ending address, again relative to the base address of the compilation unit referencing this location list. It marks the first address past the end of the address range over which the location is valid.
>
> 3. A location expression describing the location of the object over the range specified by the beginning and end addresses.
>
> [...] If all of the address ranges in a given location list do not collectively cover the entire range over which the object in question is defined, it is assumed that the object is not available for the portion of the range that is not covered.

It is quite evident how the information described above can be used to reconstruct the content of the automatic variables area in every allocation slot, and in particular the different possible frame variants depending on the address within the compiled routine. Such information can then be refined, as previously discussed, in order to increase the accuracy in determining the address ranges in which the pointers contained in the automatic variables area can be alive.

6.8 Blocks obtained from "`alloca()`"

Among the countless system routines available to Unix-derivatives, a particular system call is available on several systems to allocate a chunk of memory directly on the stack. The call `void *alloca(size_t size)` has BSD origins, and operates in a manner somewhat similar to `malloc()`. The function is intended to allocate the block of memory on the stack, and free it automatically when returning from the routine which performed the allocation. The call is of interest because the allocated memory block could be used to store pointers, and we need to discover them as part of the stack analysis.

While its intended functionality is interesting, `alloca()` has a number of problems. For instance, there is no clean way of detecting an error condition from `alloca()`, which may lead to stack corruption. Additionally `alloca()` is not fully portable, since it involves adjustments of the stack pointers out of the ordinary stack handling done by prologue and epilogue. To try to support the routine with fewer problems, and on more machines, there are even `malloc()`-based implementations of `alloca()`, in which the memory is actually allocated on the heap. In general, however, the use of `alloca()` is almost invariably discouraged.

Apart from those considerations, our only concern is the availability of a mechanism for discovering the pointers in the newly allocated block. In general, that cannot be done because of the lack of information about the use that will be made of that memory block. Nonetheless, it is reasonable to imagine a customised alternative to `alloca()`, working in a similar way, in which a descriptor of the memory block layout is passed as a parameter, rather than the size. A reference to the descriptor could then be copied to a convenient position on the allocation frame, so that, even at runtime, it is still possible to discover the pointers contained in the allocated block. The additional complexity, however, hardly justifies the support for a scarcely-used routine, whose effect can be achieved in other ways. Some support for that kind of functionality could nonetheless, at least in principle, be offered.

6.9 Registers save area

As explained in Section 4.4, during the prologue and the epilogue the values of some registers may be saved and restored, respectively, in order to make those registers available to the called routine. The scheme that is followed, to a first approximation, is presented in Figure 6.9.1. A gray box means that the mode for the corresponding register has the same mode in which it was set by the caller. When a routine is called, a new activation frame is created and the instructions of the prologue save the content of some registers in memory, in predetermined locations in the new

frame. Alternatively, the prologue could save some registers in special additional registers. After their values are saved, the registers are represented with a white box in the diagram, meaning that they now are available for exclusive use by the current routine. They may therefore be, depending on the code, either ready to be used, currently used, or no longer used. After the body execution is complete, the previous values are restored by the instructions of the epilogue. As previously mentioned in the case of return addresses, even if register windows or similar devices are used by the microprocessor, some space should really be reserved in each allocation record to store the saved values of the registers. If register windows are used, a system call is usually available to flush the content of the hidden registers to the corresponding frame locations.

As a consequence, while the registers become unused and available, following the same pattern the stack locations reserved in the save area for the registers become “alive”, with the mode that the corresponding register had in the caller. The save area remains active for the entire duration of the routine (except if the routine never returns, of course) and it becomes inactive again while the values are copied again from the frame to the registers. In many architectures the availability of specialised instructions means that several registers, possibly all of those that need to be saved, can be transferred in a single step.

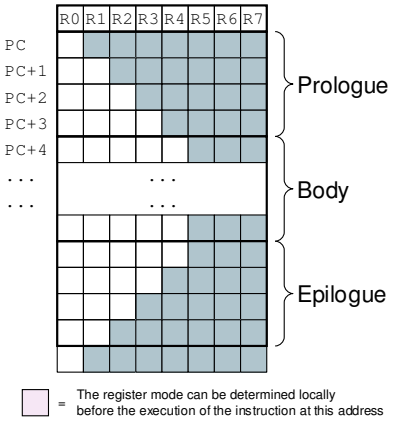


Figure 6.9.1: Local registers in prologue, body and epilogue

Discovering pointers in the registers save area can be easily done, using the same kind of information already generated in prologue and epilogue to discover the mode of registers. It is trivial, while the code for prologue and epilogue is being generated, to generate descriptors that pinpoint which frame slots become alive at every step. At runtime, when it is necessary to discover the pointers in the save area, it is sufficient to check the current program counter against those descriptors, in order to reconstruct which locations in the save area are currently parking the values of saved registers. Checking the mode tables of the caller, looking up the modes corresponding to the current return address, is enough to determine which of the active locations in the save area contain pointers.

6.10 Temporary values

Occasionally, during the compilation of particularly long and complex expressions, the compiler might run out of temporary registers to store the intermediate results of the subexpressions. In that case, some additional space is used on the stack in order to park the intermediate results that cannot be stored in the available registers. While that is a fairly common event with those microprocessors which have a limited number of registers (such as the x86 family), it is more rare on architectures with a larger number of registers, but it may happen in both cases.

The way in which some stack space is used to store temporary values is basically the same as has been discussed for outgoing arguments in Section 6.5. If the architecture in use has PUSH and POP operations, the temporary values are generally deposited on the stack and removed from there at the end of the evaluation of the expression. If, on the other hand, PUSH and POP are not available, the size of the area necessary to store all of the temporary values, which can be pre-calculated statically, is added to the size of the activation record for that routine, so that a number of locations, accessed using an offset from either the stack pointer or the frame pointer, are available to store the temporary values. Exactly in the same way that was discussed for arguments in Section 6.5, it is possible, in this case as well, to reconstruct the life of the locations used to store temporary values, and most importantly their modes, which allows us to discover the pointers during runtime.

6.11 Objects

Analysing a program written in an object-oriented language, it may be possible to determine that certain objects can actually be allocated on the stack, saving the overheads imposed by heap management and garbage collection [GS98, GS00]. If such an object is present on the stack, it could contain pointers that must be taken into account during pointer discovery, at runtime.

It is reasonable to assume that, if the object is allocated as part of the activation frame, its size and structure are known statically. In that case, the object can be simply treated as any other record, with a fixed structure. Even in the unlikely case that the object is allocated on the stack but its structure is not statically known, the object can be still treated as a kind of semi-dynamic variable. If the structure of the object is not known statically, there must be a way to discover that information at runtime. A reference to some sort of descriptor will be therefore available from the object, so that the message dispatcher can find out which method should be called. Such information, therefore, will also be available to the runtime component of the pointer discovery, giving it a key to find out dynamically the layout of the object, and finally to discover its pointers.

6.12 Other information on the stack

Sometimes, data that does not easily fit in one of the previous categories might be present on the stack. Dealing with such special cases may or may not be feasible, depending on the specific case and on the implementation details. Although the discussion will not go into much detail, the technique of trampolines will be introduced as an example.

The technique of trampolines was introduced by a paper by Breuel appeared in USENIX-88 [Bre88], and it can be described as a mechanism to support lexical nesting and lexical closures in C++ while preserving the existing function calling conventions. The core idea is creating a short segment of code directly *on the stack*, and jumping to it. The technique is actually rather useful, and it is actively used by GCC.

The trick used by trampolines relies on fine implementation details and, most importantly, it requires that the stack data area is executable (which may be unfeasible, or strongly discouraged, on

some architectures). If the microprocessor has separate caches for instructions and data, writing in the stack might not cause the instruction cache to be updated, and special precautions must be taken. The presence of a trampoline does not involve, in itself, pointers to the heap, but the code fragment, or the registers it uses, might contain pointers to code and to the stack, which must be found if the stack or the executable code are to be moved. While it is likely that trampolines can be supported, albeit with some effort, their presence is an example of the kind of unexpected difficulties that might be present while discovering pointers, and that require a very strict integration with the implementation techniques used by the compiler.

"What time is it?"
"I don't know, it keeps changing."
— *Anonymous*

Chapter 7

Pointer Discovery in the Heap

7.1 Pointer discovery

The discussion has centered, so far, on procedures and techniques that can be used to discover pointers in the registers and in the stack. These pointers form a set of roots, which can be used to determine which heap blocks are directly accessible from outside the heap (globals, as previously said, do not present particular implementation problems, and are not discussed in detail in this analysis). If we intend to manipulate the heap, for instance in order to perform garbage collection, compaction, or to copy the heap content somewhere else, it is also important to know which heap blocks can be reached following pointers contained in other heap blocks, so that it is possible to determine the set of all the blocks transitively reachable from the roots, and therefore alive according to reachability.[1] It should be noted, incidentally, that moving a heap block can in general be quite an expensive operation, since it involves finding and adjusting all the live pointers to that block.

In order to reconstruct the possible paths that can lead to a heap block, it is necessary to discover the locations, in the memory blocks contained in the heap, of all the possible pointers. In general, it is impossible to predetermine statically whether at a certain stage during execution one of those locations will really contain a valid and live pointer, so the values contained should be treated conservatively. However, since it is possible to exclude the presence of pointers in the remaining memory locations in heap blocks, the set of objects potentially reachable can be calculated, conservatively, assuming that all the pointers found are valid and alive. Furthermore, since the locations reserved for pointers can never contain valid scalar data, even values that appear to be valid pointers, even though they are actually random data, can be freely modified.

[1] There are other criteria that might be used to determine that certain objects are no longer alive. For instance, a reachability-based analysis could find that a certain block is still reachable, but a flow analysis might be able to determine, instead, that the block will never be used again.

7.1.1 Block layouts

Knowing the possible locations of pointers in heap blocks could be accomplished dynamically using tagging techniques, but that would involve a substantial overhead during execution, or would require specialised hardware. If as little overhead as possible is desired during the execution of compiled code, a simple alternative is requiring that the layout of each memory block is supplied to the heap manager during allocation, using some sort of descriptor. The heap manager will then link the descriptor, or a private copy of the same, to the newly allocated block. At runtime, when it is necessary to discover pointers in the heap blocks, it is then trivial to inspect the layout descriptors associated to each block.

A drawback of this approach is the inability to support the conventional C allocation routines, which only specify the size of the block when allocating, as in the following prototype: `void *malloc(size_t nbytes)`. The use of those routines in existing C programs would therefore need to be changed to add a reference to a descriptor, informing the system of the intended use of the block which is being allocated. In most other languages, the allocation takes place specifying the type of the entity which is being allocated, rather than its size, which simplifies the automatic extraction of the layout and does not require modifications of the source code in order to obtain the functionality required.

It should be noted that linking a block layout to each memory block implies a space overhead of one additional pointer per block in the heap. In object-oriented systems, the average size of allocated objects is rather small and the total overhead could be significant. In that case, if the object-oriented language is class-based, it may be convenient to associate the block layout descriptor to the class rather than the individual objects, so that the pointer to the object class can also be used to determine the object layout. That solution only requires additional data to be added to the classes, and there is no space overhead for every individual object.

The layout of the memory block can be either specified manually or extracted automatically, if enough information is available. An easy way to obtain the layout information relevant for specific record types (or object types) is the use of debugging information, which can act as a standard interface to access the internals of the compiler. Such an approach was used, for instance, by Kakkad, Johnstone, and Wilson [KJW98], who describe how it is possible to obtain type descriptor records (describing the full low-level layout, and not just the location of pointers) from debugging information, using the infrastructure provided by the GNU debugger, `gdb`.

The ability to distinguish statically pointers from non-pointers relies on the assumption that heap blocks never change their layout after allocation. Variant records, for instance, would cause more than one layout to be possibly associated with a heap block and, at runtime, it would not be possible to know which of the variants is in use, at least in the version without discriminant. The ability to discover pointers would then be severely restricted. A possibility could be using a conservative analysis (losing the ability to move memory in the heap) considering a combined layout which includes all of the possible locations used for pointers in a given block. An alternative could be to reorganise the layouts so that no location is ever shared between pointers and scalar values, similarly to what was previously discussed for stack frames in Section 6.7. This alternative, fairly simple to implement, could however prevent interoperability with existing code.

The use of a discriminant (also known as tag, or selector) for the variant record allows the runtime system to distinguish dynamically which of the possible layouts is in use at any given moment. However, this solution could be somewhat complex to implement, given the need to reconstruct in the runtime system the binding between the values of the discriminant and the possible layouts of the variant record.

Another case in which the layout of a memory block, used as an object, could change dynamically is, at least in principle, the use of an object-oriented language that has the ability to mutate the structure of its objects dynamically. There is however some freedom while implementing such a feature. If the size of the block needs to be changed, for example, a trivial solution would be to allocate a new heap block and copy the relevant parts of the old content, rather than rearranging the heap structures to accommodate the new size. The important aspect of altering an existing block, in any case, is the need to cooperate with the runtime system and the heap handler so that an incoming request for a service routine can be dealt with properly when parts of the heap are undergoing changes, since the layout descriptors associated to one or more objects can be temporarily invalid.

7.1.2 Allocation

Once the layout of a memory block is determined, at compile time in principle (but at any time before allocation is sufficient), the possible locations that can contain pointers can be discovered at runtime. However, there are two more details about the block creation that should be discussed: what happens if a service routine is requested while a new block is being allocated, and whether a block should be initialised.

Allocating objects in the heap is, by definition, an operation that changes the structure of the heap. If a service routine alters preemptively the heap while an allocation is taking place a conflict might arise, and the heap might be left in an inconsistent state. This is a common problems in heaps shared by multiple threads which are scheduled preemptively, and similar solutions can be adopted in this case. The obvious solution is considering the core portion of the allocation atomic. A useful paper by Shivers et al. [SCM99] discusses the issues surrounding atomic heap transactions, and the possibility of aborting a partially completed heap allocation, rather than completing it, should an interrupt arrive during the sequence. Similar techniques can be adopted in our case.

7.1.3 Initialisation

Once the heap structure is modified, the newly allocated block still contains random data. The choice, at this point, is whether to initialise the content of the block to some standard value (either the whole block or just the pointers) or, alternatively, leave the block as it is. There are two factors that concur to the decision. Some programming languages, like Java, assume that the content of every new heap block is entirely initialised to standard values (in Java zero, false, null, and so on). In those languages, it would make little sense to return a block containing random values and leaving to the calling code the task of initialising it. For other languages, like C++, the initial content of the memory block has no relevance. A second factor is the accuracy that we wish

to achieve on the pointers. Preinitialising all the pointers to a standard value, null for instance, before the block can be used by the language, ensures that the pointers that are initially unused are recognised as such, rather than being treated conservatively. There is a trade-off between the additional time spent initialising the block (or at least its pointers) and the potential future savings in terms of memory and time, due to lower number of possibly invalid pointers treated conservatively.

If the memory block is to be initialised before control is returned, there is an additional detail concerning interrupts which might be received during the initialisation stage. If a service routine, and consequently a pointer discovery, are requested while initialising, it might be useful to inform the runtime module that there are really no valid pointers in the block yet, and that the values currently present, initialised or otherwise, should be ignored. That can be easily accomplished leaving the reference, which should refer to the layout descriptor, initially set to a standard value, for instance null, during the initial allocation stage. If a pointer discovery is requested, the runtime will detect the standard value and skip the block altogether. Once the initialisation stage is complete, the real value of the reference to the layout descriptor can be moved atomically in the location corresponding to the new block. Every pointer discovery request received immediately afterwards will find all the pointers properly initialised, and easily recognisable as not currently in use.

7.1.4 Code in the heap

A final note should be made about storing code in the heap. During the discussions about pointer discovery in the registers and the stack, a number of considerations were made about pointers that are guaranteed to be pointing somewhere within the stack, or within the code, as opposed to heap pointers. If some code can be stored in a heap block, however, the situation becomes somewhat more complex when block relocation is required. If code residing on the heap is called, and that code in turns calls other code, part of the dynamic chain will contain addresses that refer to locations within heap blocks. If such blocks are relocated, the dynamic chain needs to be adjusted, as well as potentially all the pointers which can be used to refer to code. The runtime, and the data structures that are prepared for use by the runtime module should reflect the new possibility.

An additional factor is that the pointers will not, in general, point to the base address of the heap block any longer. When a heap block is moved, it is necessary to adjust the value of all of the pointers which refer to that block, including possible pointers to code within the block. In that case the pointer to the base of the memory block should be calculated from the value of the pointer that refers to the code, so that the pointer can be updated if the block is moved. More details about this aspect are available in Chapter 10, devoted to derived pointers.

Being unable to separate pointers to heap blocks from pointers to the code also requires more operations while performing certain operations. For instance, we have previously seen that certain locations on the stack are always used to refer to code locations and as such can be ignored while calculating the reachable set of blocks in the heap. If code is contained in the heap, such an assumption is no longer valid. If possible, for the sake of efficiency, storing code in the heap should therefore be avoided if possible.

I wouldn't reveal the required information under torture.
But I would if bribed.
*— **DG (rgreenfield@btinternet.com)**, August 13, 2001*

Chapter 8

Runtime Module

The creation of the data structures (the PC maps) that describe the use of registers, stack, and heap, has a natural counterpart in the runtime component that will dynamically explore those structures whenever a pointer discovery is preemptively requested. This chapter will describe the overall structure of the runtime module and the general techniques that can be used to reconstruct the pointer information. Information more specifically related to the test implementation is available in Chapter 9. Before delving in the details of the runtime module, it will be useful to summarise what was described in the previous chapters, listing the information that will be available during the runtime analysis.

8.1 Data structures

A number of different data structures are created, as discussed in the previous chapters, during compilation. The exact organisation of those data structures depends on the implementation choices for the system, including the trade-off between speed of access and size, support for specific features of programming languages (packed records, semi-dynamic variables, and so on) and, of course, the target architecture. The fundamental information that will be available in those structures, leaving out the support for some less critical features, can be summarised as follows:

From the registers analysis, for each routine:

- maps that describe the mode of registers for every instruction in the routine body of the compiled routine, specifically which registers may contain valid pointers.
- information about registers used as arguments and return values, in particular which ones are used as pointers.

- the set of registers that are saved and restored in prologue and epilogue, and at which instructions each of the registers is saved/restored.
- if register windows or similar mechanisms are used, which instructions in prologue and epilogue perform the register renaming.

Also of importance are the following sets: volatile registers, call-preserved registers, and among the latter the registers actually used by each routine.

From the stack analysis, for each routine:

- which registers are used to access the various parts of the frame (frame pointer/stack pointer). Normally specified by the ABI, but may depend on leaf/non-leaf condition.
- information about where the registers are saved (which offsets in the stack frame). Such information can be omitted if the set of offsets is fixed (for instance, in the SPARC there is a fixed area devoted to that purpose).
- locations in the frame which are used for arguments and return value, and if there are any pointers among them.
- the possible layouts of locations used for automatic variables, and temporary values, associating the instructions in the compiled body to the offsets, or range of offsets, of locations reserved for pointers. That includes, if supported, information about semi-dynamic areas.
- if required, information about further locations, in the stack frame, that may contain pointers to the heap.

From the allocations of heap blocks:

- layout of each block, with the position of pointers. This information is not necessarily available statically, but when a pointer discovery is requested, a layout descriptor will be available for each heap block.

All of the above information will be available to the runtime module, described in more detail in the following section.

8.2 Structure of the runtime module

At runtime, the program compiled by the customised compiler will require some support for its execution, and for the proper handling of service routines that may need to inspect, or alter, the heap content. The necessary infrastructure will be offered by the runtime module, which, among its functions, will coordinate the use of the heap between the program and the service routines, and discover the pointers when a preemptive request is received. The exact design of runtime module and heap manager, of course, are specific to the implementation. Overall, however, the

runtime module will probably have a structure resembling the logical diagram in Figure 8.2.1, which is also mostly followed by the test implementation detailed in Chapter 9.

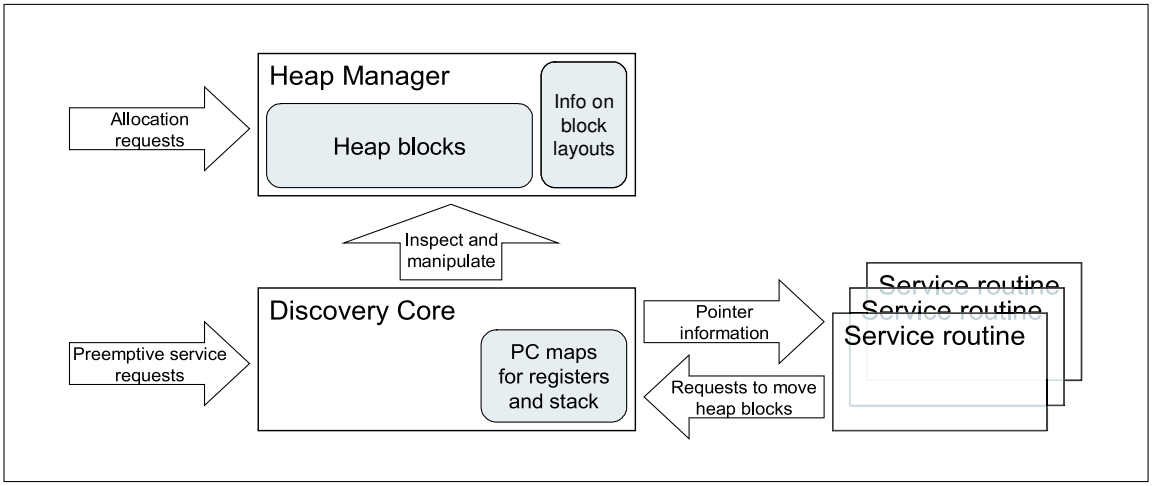


Figure 8.2.1: Structure of the runtime module

The runtime module revolves around a discovery core, interfaced with the heap manager and pluggable service routines. The heap manager deals with allocation requests from the program, and provides an interface through which the discovery code can inspect and manipulate the heap. In particular, the heap is inspected by the discovery core in order to find pointers, and it is modified by the discovery core whenever a heap block should be moved. Every time the program requests a heap allocation, an associated descriptor is passed to the heap manager (which might make a local copy of it) to specify the block layout.

The discovery core assumes control preemptively, temporarily suspending the normal execution. The state of the suspended thread, including the content of user-accessible registers and other system information, is saved by the system in its internal structures, in order to resume normal execution later. On many systems, such information is accessible using a standard structure, known as a context, which also contains details about stack size and location. Using the mechanisms that will be shortly discussed, the discovery core inspects the saved state of the registers (including the program counter), the stack, and the heap, using the information contained in the data structure statically created by the compiler for registers and stack, and the descriptors for heap blocks. The set of locations that can contain pointers is determined, and it is passed to the service routine.

The service routine uses the set of pointers calculated by the discovery core to explore the heap, and read its content. If a memory block is to be moved, the service routine asks the core to alter the heap appropriately, maintaining the necessary bookkeeping information. Although not shown in the diagram, the core will modify the stack and the saved registers whenever necessary in order to reflect the new location of the memory blocks. Once the service routine has performed the heap manipulation required, control is returned to the discovery core, which performs the final clean-up and updates the context to reflect the modified values of pointers contained in registers

and stack. Control is then returned to the user program, using the updated context.

A more detailed explanation of the techniques that the discovery core may use in order to find pointers is discussed in the next sections.

8.3 Extracting the context

If a service routine needs to operate preemptively, the execution of the user program is temporarily suspended and control passes to the runtime module in order to proceed with pointer discovery. The first operation that the discovery core must perform, in order to determine the position of pointers, is finding out where the system structures related to the suspended threads are. In particular, it is necessary to find the saved state of user-accessible registers, including the program counter, and enough information about the stack of each thread to reconstruct the activation chains and the current running context. In general, that information will be made available by the system using a structure containing the "context" of the interrupted thread.

To be precise, it would be necessary to distinguish between kernel-level threads and user-level threads. For instance, the structure known as `ucontext` (user context) can be used in conjunction with the calls `getcontext()` and `setcontext()` to easily implement user-level threads in a variety of systems, including Linux, Solaris, OpenBSD, NetBSD, and AIX. The calls `getcontext()` and `setcontext()`, according to the manual pages of NetBSD 1.6, first appeared in AT&T System V.4 UNIX. In the systems conforming to the Open Group Base Specifications [Ope03], the `ucontext` structure can also be obtained automatically as part of the signal handler invocation, in which case the structure contains a great deal of information about the suspended thread, including the value of user-level registers, the size and location of the stack, and so on. Using `ucontext` structures, it is quite easy to implement user-level threads, controlling manually the context which corresponds to every thread, which simplifies the task of finding the whole set of contexts whenever a signal is received.

Using standard thread libraries it might be more tricky to suspend all the threads, and to extract the context information. The specific approach will depend on the thread model in use, since they vary in the set of calls and conventions available. The most common thread models are the popular ANSI/IEEE POSIX 1003.1c-1995, DCE threads, Solaris threads (also known as Unix International threads), and Microsoft-style Win32 threads. Suspending arbitrary threads can be accomplished directly in the Solaris-style model, using the calls `thr_suspend()` and `thr_continue()`, and in Win32 using `SuspendThread()` and `ResumeThread()`, but no equivalent primitives exist in POSIX threads, which may require some more effort or less portable techniques (for instance, sending `SIGSTOP` and `SIGCONT` using `pthread_kill()` may work on some systems).

Regardless of the specific technique used, and the gory implementation details, what is necessary is the ability to determine the set of threads we wish to operate on, the ability to suspend them, and the context information for each. Once the stack location, program counter and other registers are known, it is possible to start inspecting the statically built tables and the heap layout descriptors to find pointers. The operation can be subdivided into simpler steps, corresponding to

logically distinct storage areas, as will be explained in the following section.

Before discussing the details of pointer discovery, however, it is important to point out an issue that might derive from the preemptive suspension of threads. So far, it has been implicitly assumed that, whenever a thread is preemptively interrupted, the code which is being executed in that moment, having being compiled by the customised compiler, has associated to it the set of data structures necessary to discover all the pointers. While such an assumption may be true in the case of a system entirely developed using the customised compiler, there are cases in which that might not be the case. If the code generated using the customised compiler, for instance, is hosted by a traditional system and makes use of existing libraries, an interrupt could be received while a thread is in code with no associated data structures, making it impossible to fully determine whether pointers are contained in registers and stack frames at that moment. Another case in which a preemptive request for a service routine could not be served immediately is the case in which, for whatever reason, a critical section of code needs to be completed before anything else can happen, for instance a section of code which is manipulating the heap structure. In all those cases, it is necessary to make sure that control is returned, for each thread, to "safe" code, in which every instruction can be treated as a safe point. While a full explanation of the issues involved will be presented in Section 8.5, the following section will focus on the pointer discovery procedure for a single thread, assuming it to be suspended at an instruction in the custom code for which the additional data structures are available, and not within a critical section.

8.4 Pointer discovery

The pointer discovery operation can be subdivided into a number of simpler tasks, depending on the content of registers, stack and heap. Although the specific details may vary with the implementation, it is possible to distinguish three separate sections:

- user-level registers and register save areas in the stack
- automatic variables, temporaries, arguments and return values saved on the stack
- heap blocks

The operations that need to be performed for each section are now examined.

8.4.1 Registers and register save areas

When a request for pointer discovery is received, the content of the user-level registers is available in the representation in memory of the context, as previously explained. Referring to the diagram in Figure 8.2.1, the value of the program counter can be extracted from the context, and used to look up the use of registers for that instruction, by checking the maps associated with that instruction. However, as we have seen, the data structures describing the modes of registers is built relying exclusively on information that can be obtained locally, inspecting a single routine. In the case in which a register has a certain mode prior to entering a routine, and is not used at

all until the end of the same routine, there is no information in the map of that routine which can be used determine what the mode of that register should be. In that case, the maps related to the calling routines must be examined. Furthermore, while prologue and epilogue are being executed, the mode of a certain register may depend on the local maps or on the maps of the caller, depending on whether the previous value of that register has been saved or not at a given instruction.

In order to clarify the mechanisms, and to make the discussion more systematic, let us recall the categories of registers introduced in Section 4.4. The user-level registers are grouped in four categories: purpose-specific registers, global registers, volatile registers, and call-preserved registers. Many purpose-specific registers, because of the way they are used by the compiler, will never contain a valid pointer to a heap block. That may include registers that could actually be used as general purpose registers, according to the microprocessor architecture, but that are used by the compiler exclusively for specific tasks. For instance, as long as we are only interested in relocating heap memory, we never need to look for pointers to such memory in the program counter, the stack pointer, the frame pointer, special registers used as loop indexes, as static chain pointer, and so on. The purpose-specific registers and the global registers that contain pointers to the heap will be known statically, and determining their mode does not involve any particular problem.

The remaining user-accessible registers can be divided into two categories: registers that are assumed to be volatile across calls, and registers that must have their value preserved across calls. If the caller needs to make sure that the content of one of the volatile registers survives across a call, it is the caller's responsibility to save its content somewhere (either in another register or in memory) prior to the call, and restore the value afterwards. Symmetrically, if the callee wants to use one of the call-preserved registers, it will be the callee's duty to save the content of the register during the prologue, and restore the register in the epilogue, before returning. The separation between volatile and call-preserved is fixed and dictated by the ABI, and it needs to be unique in order to ensure interoperability between code generated by different compilers.

This simple distinction is enough to give a preliminary indication of where is is possible to find the mode information for a certain register. If the register is volatile, it is by definition the "property" of the more recently activated routine, and its mode information must be available in the corresponding mode map. If a volatile register is not found in the map, then it is unused. On the other hand, if a register is call-preserved, the register could either be used by the more recently activated routine, or by the caller (or one of the caller's callers).

In the case of call-preserved registers, it is necessary to check the map corresponding to the value of the program counter saved in the current context. If the register is present in such map, and the routine was interrupted in the body, then the register is owned by the current routine, and its mode can be determined immediately. If the register is present in the map, but the routine was interrupted in prologue or epilogue, then the position of the program counter should be compared with the information about registers saved/restored contained in the related descriptors. If the register has already been saved, or has not yet been restored, then it is owned by the current routine and the mode can be found in the local maps. If the register has not yet been saved, or has already been restored, then the mode of the register depends on the caller, and the register map of

the latter will have to be checked. Finally, if the register is not at all present in the register map for the current routine, then the mode depends once again on the caller.

For those registers whose mode depends on the caller, the topmost return address must be considered, leading to the value that the program counter had during the execution of the call instruction. Such value of the program counter can be used to check the register maps of the caller. If the register is found, the mode can be determined. Otherwise we are once again the case in which we cannot tell immediately what the mode is, but this time we cannot be in prologue or epilogue. Therefore the register is not used by the caller, and it is necessary to keep inspecting return addresses from the dynamic call chain until either the register is found in one of the maps or the stack has been fully explored.

Here is a summary of the above explanation in a schematic form:

- for volatile registers, check map for current PC:
  - if found, the mode can be obtained from the map
  - if not found, the register is unused
- for call-preserved routine, check the current PC:
  - If PC in routine body, check for register in register map:
    - if found, the mode can be obtained from the map
    - if not found, inspect the map corresponding to the return address
  - If PC is not in the routine body, check if register is saved at that value of PC:
    - if saved, check local data structures of prologue/epilogue:
      - if found, the mode can be obtained from the structures
      - if not found, register is unused
    - if not saved, inspect the map corresponding to the return address

To inspect the map corresponding to the return address:

- is the return address valid/does another frame exist?
  - if not valid/stack fully traversed, the register is unused
  - if valid, check register map for that address
    - if register found, the mode can be obtained from the map
    - if register not found, obtain the next return address and repeat

This scheme can be used to reconstruct the mode of all the registers in question. The time required, in the worst case, is proportional to the time of a lookup in the map times the stack depth. It should be pointed out, however, that the whole stack needs to be traversed anyway, to find the pointers contained in automatic variables. Also, in certain cases it is not necessary to traverse the whole stack. For the SPARC, as will be explained later, it is only ever necessary to descend two levels of stack depth to be able to fully determine the mode of all the registers.

The above stack traversal can also be combined with the stack traversals necessary for analysing the register save areas in the stack. The scheme that can be followed has similarities with the previous one. It will be assumed that, if hardware techniques like register windows are used, the content of the additional registers, used as save buffer, can be flushed to memory in corresponding register save areas. This time as well, the first operation is checking the value of the program counter in the saved context. If the value corresponds to an instruction within the routine body, it means that the register save area for the current routine has already been filled with the values that the registers had at the beginning of the prologue (except in the possible special case of leaf routines). If the value of the program counter refers to an instruction in the prologue or the epilogue, it is necessary to determine which registers are saved and which are not. Once the set of positions currently used in the register save area of the current routine has been determined, it is time to find out where the pointers are. The saved values refer to the content of the call-preserved registers before the current routine was called, therefore the mode of those locations can be obtained by checking the register map corresponding to the topmost return address for the registers corresponding to those locations in the top stack frame. If one of the registers is not found in that map, the stack should be traversed as in the case of registers, looking for the mode of the register corresponding to each used location in the register save area. Once the mode of saved registers for the current frame has been determined, the same procedure can be repeated for the caller, except that now the address will certainly be within the routine body, and the full register save area is used. The register save area contained in the stack frame of the caller can then be examined using the map corresponding to the registers as they were used by the caller's caller, using the following return address, and so on.

The above procedure can be further rearranged, so that it can be executed using a single traversal. That is done by keeping track of the stack locations, in all the previous frames, for which the mode is not determined. At every step of the traversal, if the corresponding register is found in the local map then the mode of the stack location becomes known. Also, if a register is saved by a certain routine, it is in order to reuse the same register within the body, therefore the same register also appears in the register map for the same routine. Consequently, there can be only one stack location, at every step, corresponding to one register for which the mode has not yet been determined. In order to clarify the whole procedure, an (informal) algorithmic description will be useful:

- Find the PC saved in the context. Determine the set *S* of locations in the topmost stack frame that correspond to saved registers. Each location in *S* has an associated register.
- Find the return address and consider the preceding stack frame. Check the register map for the mode information of the call-preserved registers only. For each of the registers found, check if a location in *S* is associated to that register. If so, the map supplies the mode for that location, which is removed from the set *S*. At the end, the set *S* contains the locations for which the mode has not yet been determined. Add to *S* the set of locations in the current register save area. The set being added is disjoint from *S*, since all the corresponding registers were also in the register map that was just scanned. Repeat this point, finding the new return address, until the stack is fully scanned.

- After the stack has been fully scanned, every element left in S can be marked as unused.

The scan for registers and the scan for the register save area can now be combined as follows:

- Obtain PC from the context. For all volatile registers, check map for current PC:
  - if found, the mode can be obtained from the map
  - if not found, the register is unused.
- Let C be the set of call-preserved registers. Check the current PC, and find the set of locations S used in the register save area at that position in the code. Let R be the set of registers associated to the elements in S. For all the registers in R, check the register map corresponding to the current PC:
  - if found, the mode can be obtained from the map
  - if not found, the register is unused
- Set C to $C \setminus R$
- Take the return address, and consider the preceding stack frame. Check the register map corresponding to that address, looking for the registers which are in $C \cup R$. For the registers found in the map, which are in C, the mode of those registers can be obtained from the map. Remove those registers from C. For all the registers found in the map, which are in R, the map supplies the mode for the matching locations of S. Remove those locations from the set S. After the map is analysed, add to S the set of locations in the current register save area. Let R be the set of registers associated to the elements in S. Repeat this whole point, taking successive return addresses and frames, until the stack is completely scanned.

The rough algorithmic description above can be substantially altered or optimised, depending on the characteristics of the microprocessor in use, the ABI specifications, and particular features or conventions in the system. Nonetheless, it should offer a basic idea of the way in which pointers can be found in registers and the register save areas. The time complexity is order the time used to explore the register maps multiplied by the number of locations in the save areas on the stack.

8.4.2 Stack and heap

As shown in the previous section, finding pointers in registers and register save areas can be rather complex, but can be accomplished by reconstructing the way in which pointers are successively saved in frame locations by the various routines across nested calls. Determining where pointers are in the remaining portions of the frames, and in the heap, is considerably easier, although some attention must be paid to those portions of the stack frames that act as an interface between different routines, specifically arguments and return values. In that case, a mode indication exists in both the maps of the caller and the callee, as also explained in Section 6.5. For instance, let us assume that the caller is passing an argument to the callee, using a memory location on the stack. While the argument is being prepared by the caller, the mode for the corresponding location can be obtained by checking the maps associated with the caller. The call instruction is

then encountered, and execution continues in the callee. The mode for that very same location can now be determined by checking the maps of the callee.

That leads us to a simple scheme that can be used to discover pointers in the parts of the frames used by arguments. When a pointer discovery is requested, the PC saved in the context is used to find the most recently activated routine. For that routine, the associated maps can be used to determine the mode of both the incoming and the outgoing arguments (if any), in the topmost stack frame. Once that is done, the previous return addresses are considered in succession. For each of them, the maps corresponding to each return address can be used to determine the mode, in the corresponding frame, of the incoming arguments only, since the outgoing arguments have already been analysed as incoming arguments for the previous level. Very similar considerations can be made for the locations reserved for storing return values.

Apart from the cases of arguments and return values, the remaining portions of each stack frame, described exhaustively in Chapter 6, can be analysed by considering the maps of a single routine, in isolation. Even if an automatic variable is modified by a nested routine, for example, the mode of the associated location, in a statically typed language, will not change. A pointer to some sort of object, for instance, will remain a pointer regardless of the specific object which is being pointed to. Discovering the mode of stack-allocated automatic variables, temporaries and other frame components is therefore fairly easy.

Finally, discovering pointers contained in heap blocks is trivial, if the heap manager maintains the association between each heap block and the descriptor that was specified at allocation time. Finding the pointers is therefore just a matter of scanning the descriptors for each block. The pointer information so obtained about the heap, and the one obtained about the various parts of the stack frames and the registers, can then finally be passed to the service routine, which will perform the heap manipulations it requires.

8.5 Critical sections and foreign code

Even though PC maps can support, in theory, preemption at any program location, practical issues might limit this ability. Certain portions of code, for instance inlined heap allocations (see Shivers et al. [SCM99] for useful information), might need to be atomic (“critical sections”). Using write barriers, while using generational GC, presents similar problems: moving a heap block between a card marking and the actual write will almost certainly cause problems. Critical sections can be dealt with fairly easily using software or hardware breakpoints. If an interrupt is received while the program counter is within a critical section, a breakpoint can be inserted on-the-fly right after the critical section, and control returned. Other techniques are also possible, like completing the critical section manually, for example using emulation, step-by-step execution, or running some external code that performs the same function.

A more complex problem can arise when linking code generated by the customised compiler and “foreign” code compiled with a standard compiler, and that has no maps associated with it (for instance system code or generic libraries). If the microprocessor is interrupted in such foreign code, there might be not enough information to determine the location of all the pointers

currently in use. If code with and without PC maps can be arbitrarily mixed, the presence of foreign stack frames might cause problems while reconstructing the pointer information even if the microprocessor is interrupted in code for which the related information is available.

For instance, the address of a routine which has PC maps could be passed to a foreign routine, as a callback address. If the foreign routine later calls the routine whose address was passed, and an interrupt occurs during the execution of the latter, a foreign frame will be present on the stack even if the interrupted code has PC maps associated to it. It is therefore necessary to deal appropriately with those cases. The two possible alternatives are either treating the content of those foreign frames conservatively, or deferring the execution of the service routine until there are no longer foreign frames on the stack, so that an exact analysis can be performed.

Fisher and Reppy [FR] rely on a partially conservative analysis, treating all foreign values as possible pointers. Their GC is mostly-copying. The choice of Stichnoth et al. [SLC99], instead, is to return control immediately to the running thread and retry after a while. If, in the meantime, the thread reaches a point known to be GC-safe, such as an allocation or a synchronisation routine, GC can proceed immediately. Such an approach, while effective in practice, might needlessly prolong the latency before the service routine is executed.

A possible alternative, implemented in the prototype discussed later, is to scan down the call chain looking for a return address which guarantees the ability to reconstruct pointer information. This technique was used by Moss and Kohler in the Trellis/Owl system [MK87]. In practice, this amounts to looking for a group of contiguous stack frames corresponding to code with PC maps, without foreign stack frames in between. The return address that would reactivate the topmost stack frame in that group is then saved and replaced with a custom handler, and control is returned. Execution can then continue at full speed through the remaining foreign code until the microprocessor is about to return to code produced by the customised compiler, at which point the control flow is automatically intercepted by the runtime module, thanks to the patched return address. Finding the best return address to patch, in practice, can be rather difficult considering that part of the dynamic call chain can be in kept in registers, according to the peculiar conventions of the specific microprocessor. The case of the SPARC is particularly interesting in this regard, and more details about this aspect are available in Section 9.7.1.

As a particular case, if a foreign routine stores a pointer to a heap block in its global area (a C static variable, for instance) and returns, such a pointer will be, as far as the runtime system is concerned, invisible. That might lead to unexpected consequences, like the premature release of a memory block still in use. A limitation that must be enforced while using pre-existing code, therefore, is that the called foreign code must not privately store copies of pointers to heap blocks passed as parameters.

Continuing execution in foreign code, until an instruction “safe” enough to perform GC or similar operations is encountered, is a viable technique if a single thread is in use. However, great care must be taken if multiple threads are simultaneously active, and each of them can run foreign code at any time. In that case, continuing execution in one of the threads might cause a situation of deadlock if the resumed thread requires resources that are held by one of the suspended threads. A careful analysis is required in that case, with the possible addition of supplemental synchronisations or particular scheduling techniques in order to prevent deadlocks.

It should be underscored that the issue only arises when mixing together code generated with the customised compiler and foreign code. Having suitably adapted libraries, compiled with the customised compiler, would allow every instruction in the libraries to serve as a safe point as well, and support for multithreaded applications would be straightforward.

The simple things you see are all complicated.
*— **Peter Townshend**, lyrics in "Substitute", 1966*

Chapter 9

Implementation

In order to expose the hidden technical challenges that might be faced by those wishing to implement a system like the one described, an experimental implementation was created using GCC. A notable characteristic of the test implementation is that only the compiler back end was modified, which allowed the system to be tested using sample programs written in multiple languages. In particular, C, Pascal and Ada were used to write test programs, and PC maps were also created, with some limitations, for C++ and Java code. It is the first time that a system able to generate PC maps for multiple languages is described in the literature.

The test implementation was created by customising the back end of recent versions of GCC (development started with version 2.95.2 and continued up to version 3.3.3). The output of the customised compiler was then postprocessed, assembled, and linked with the runtime module, which offers the required interface with a sample service routine. A complete diagram of the system will be presented shortly, together with a detailed description of the most relevant implementation techniques used in the project. In order to introduce properly the working environment, the GCC compiler suite will be now briefly introduced, with particular regard to the aspects relevant to the creation of PC maps.

9.1 GCC in brief

GCC (GNU Compiler Collection) is a very large and sophisticated piece of software, which comprises, in recent distributions, something in the region of three million lines of code, excluding optional front ends like Pascal and COBOL. The resulting compilation infrastructure is able to generate code for several different microprocessors, conforming to the calling conventions of multiple operating systems, from a number of high-level languages. It is even possible to create on a given platform X an executable version of GCC that will run on a different platform Y and that will produce code for a third platform Z.

It would be impossible to achieve this flexibility without a careful modularisation of the whole system. GCC employs separate description files for each front-end, back-end and operating system, in order to produce a separate compiler for each triplet. Each of those compilers works, from a logical point of view, according to the simplified diagram shown in figure 9.1.1.

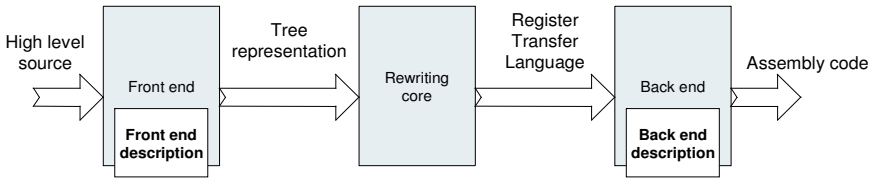


Figure 9.1.1: Stages of compilation in a GCC compiler

The high-level intermediate representation, a set of tree data structures, contains a source-language-independent description of the high-level program. The original source file is transformed into an abstract form that still refers to complex types, records, arrays, case statements and so on, but in a way that is no longer tied to the source language. The high-level intermediate representation is then transformed into a low-level register transfer form, in which the high-level constructs are translated into simpler forms that use primitive data (integers, floating points, pointers) and simple operations, such as basic arithmetic, tests and jumps. The register transfer representation is then optimised and reorganized multiple times, according to the microprocessor characteristics, until the final assembler code is generated.

The front-end and the back-end perform the relevant transformations according to a modular description of the source language and the target microprocessor architecture, respectively. Those descriptions are composed of a mixture of macro definitions, fragments of C and specialized data files. The interface between GCC and the descriptions is cleverly designed to keep into account almost every conceivable feature of modern languages and microprocessors, so that new, even if very unusual, languages and microprocessors can be supported just by designing a suitable description. In particular, the main part of a back-end specification can usually be implemented by writing just three files, totalling about 5,000-20,000 lines, depending on the complexity of the architecture.

9.1.1 GCC and the Register Transfer Language

The internal low-level representation of the compiled program is contained in data structures that can be handled using a set of specialised functions and macros. This internal representation is normally never fully translated into a human-readable form. However, for debugging purposes, it is possible to obtain a diagnostic dump of most of the structures, which are displayed in the form of a list of LISP-like expressions. A similar form is also used in one of the files of the back-end description, and it is converted into internal structures only once, when the compiler is generated. Describing in full detail the register transfer language is beyond the scope of this thesis, but some examples will be necessary in order to clarify the techniques used.

Let us consider the following fragment of C code:

```
void abc() {
 long  a=0x2ba76dc2;
 short b=0x47bd;
 char  c=0x5e;
}
```

This is the RTL dump of the three assignments during the initial stages of compilation (on the x86 architecture):

```
(insn 9 6 12 (set (mem/f:SI (plus:SI (reg:SI 38 virtual-stack-vars)
                (const_int -4 [0xfffffffc])) 0)
        (const_int 732392898 [0x2ba76dc2])) -1 (nil)
    (nil))
(insn 12 9 15 (set (mem/f:HI (plus:SI (reg:SI 38 virtual-stack-vars)
                (const_int -6 [0xfffffffa])) 0)
        (const_int 18365 [0x47bd])) -1 (nil)
    (nil))
(insn 15 12 18 (set (mem/f:QI (plus:SI (reg:SI 38 virtual-stack-vars)
                (const_int -7 [0xfffffff9])) 0)
        (const_int 94 [0x5e])) -1 (nil)
    (nil))
```

The translated code seems quite complex, but it can be easily by explained analysing the various nested subexpressions. The shown expressions are organized in a doubly linked list of instructions (“insn”). The first of the three numbers is an identification number assigned to the current insn, the second one is the insn preceding and the third one is the insn that follows in the chain. Each of the three instructions is an assignment (“set”) to a memory location (“mem” subexpression) of a constant (“const_int 732392898”, for example). The memory access subexpression is obtained by adding to a base pointer, contained in the virtual register number 38, an offset (const_int -4, for example, in the first insn).

A crucial aspect of that RTL code is the existence of well-defined low-level primitive types (SI-Single Integer for 32-bit, HI-Half Integer for 16-bit, QI-Quarter Integer for 8-bit and so on), that are used by the back-end to find the most appropriate expansion for each of the various subexpressions during the various passes, until the final assembler code is generated. RTL defines, and is able to handle, a variety of primitive types. The available ones include integers of various sizes, from a single bit to very long 32-byte integers, various sizes of floating points (currently up to quadruple-precision), arbitrary memory blocks and even complex numbers, just in case the target architecture offers hardware support for them.

However, not all of these primitive types need to be available or defined in the target machine code. If a certain data size is not available, GCC will automatically restructure the operations in order to use the available hardware characteristics of the target microprocessor. For instance, if the back-end description informs GCC that the microprocessor is able to perform a SI (32-bit) sum directly, the corresponding code will be used, otherwise the SI sum will be automatically split into multiple 16-bits sums, and so on. In that way, advanced features of each microprocessor can be used in an optimal way, while less powerful microprocessors can perform the same operations

using more instructions. Those low-level primitive types are called, in GCC terminology, “machine modes”. It will be our task to convert those machine modes (SI, HI, QI, etc.) to the modes necessary to perform the memory manipulations required by the system routine. In the cases of garbage collection, persistence and homogeneous migration, we are interested in determining where pointers are. The modes required, therefore, will be “pointer” and “scalar”.

Within GCC, a specific machine mode (named “Pmode”) is reserved for the handling of pointers. The compiler, however, makes no assumptions on the structure of Pmode, and relies on the back end description of the target architecture to determine the concrete representation of pointers. What usually happens in practice is that Pmode is redefined, in the back end description, as an integer of a certain size (usually 32 or 16 bits) and pointers become, from that point on, just ordinary integers.

For example, in the RTL fragment already encountered:

```
(insn 9 6 12 (set (mem/f:SI (plus:SI (reg:SI 38 virtual-stack-vars)
                (const_int -4 [0xfffffffc])) 0)
        (const_int 732392898 [0x2ba76dc2])) -1 (nil)
    (nil))
```

the qualifier used for memory references is SI (32-bit), and the pointer arithmetic is performed using the same size. Distinguishing pointers without passing additional information from the front end, or customising the internals of GCC, may seem at first a bit difficult. Luckily, one of the many features of GCC can be used in unexpected ways to achieve similar results, as will be shortly explained.

9.1.2 Rule rewriting

To understand the kind of modifications that need to be applied to the back end description, it is useful to show how the main machine description file is organized.

The file “sparc.md”, one of the files that comprise the SPARC back end description, contains a series of RTL fragments, and C fragments, that define how the various passes of GCC should reorganize and optimise the generated RTL, in preparation for the final code generation. The rules are quite difficult to explain, and sometimes can be very involved. A very simple example is shown in Figure 9.1.2, on the next page.

```
(define_insn "*movsi_insn"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=r,f,r,r,r,f,m,m,d")
        (match_operand:SI 1 "input_operand"   "rI,!f,K,J,m,!m,rJ,!f,J"))]
  "(register_operand (operands[0], SImode)
    || reg_or_0_operand (operands[1], SImode))"
  "@
   mov\\t%1, %0
   fmovs\\t%1, %0
   sethi\\t%%hi(%a1), %0
   clr\\t%0
   ld\\t%1, %0
   ld\\t%1, %0
   st\\t%r1, %0
   st\\t%1, %0
   fzeros\\t%0"
  [(set_attr "type" "move,fpmove,move,move,load,fpload,store,fpstore,fpmove")
   (set_attr "length" "1")])
```

Figure 9.1.2: Expansion rule in GCC

The rule "`*movsi_insn`" specifies what are the right fragments of assembly code to emit for a simple move of a 32-bit value from any place to any other place. The second and the third line of the rule (enclosed in square brackets) specify the generic pattern of the RTL expression that can be expanded by this rule. In this case, the expressions of the form "(set (operand0) (operand1))" will be considered for this rule and, if the further constraints specified are satisfied, the rule will be applied. The part inside the "`match_operand`" subexpression specifies further restrictions, and the lists of codes for the two operands help GCC to select the most convenient expansion among those listed below. For instance, to move a 32-bit value from memory to a register, the fifth expansion, `ld %1,%0`, will be selected. Each expansion can be simply a portion of text, which will be printed as part of the final assembly code, or the output of a C code fragment, that can replace the matching RTL expression with more sophisticated expressions. For example, in the following rule for the assignment of a sum between two SI values, the case in which one of the addends can be contained in a 13-bit constant can be rewritten in a special, more efficient, way:

```
(define_expand "addsi3"
  [(set (match_operand:SI 0 "register_operand" "=r,d")
        (plus:SI (match_operand:SI 1 "arith_operand" "%r,d")
                 (match_operand:SI 2 "arith_add_operand" "rI,d")))]
  ""
  "
{
  if (arith_4096_operand(operands[2], SImode))
    {
      if (GET_CODE (operands[1]) == CONST_INT)
        emit_insn (gen_movsi (operands[0],
                              GEN_INT (INTVAL (operands[1]) + 4096)));
      else
        emit_insn (gen_rtx_SET (VOIDmode, operands[0],
                                gen_rtx_MINUS (SImode, operands[1],
                                               GEN_INT(-4096))));
      DONE;
    }
}")
```

The rewriting engine used by GCC is definitely quite sophisticated and the array of techniques

used to improve the final code quality is impressive. It is possible, for instance, to specify how expansions can be split into multiple segments, so that expansions for different expressions can be interleaved in order to improve pipelines efficiency in the microprocessor. It is also possible to define which are the functional units of the microprocessor, how they work and the cost of the various expansions, so that GCC can automatically optimise the usage of the various functional units and improve the overall code efficiency. Window registers, delay slots and other possible microprocessor peculiarities are all taken into account.

9.2 The compilation process

The diagrams in Figure 9.2.1 and Figure 9.2.2 describe the compilation process in the test implementation, which follows the general discussion of the previous chapters. For each source file, as shown in Figure 9.2.1, the compiler generates, as usual, the corresponding assembly code and the optional debugging information. The customised back end, alongside the usual output, inspects the compiler internals during the final code generation stage, in order to generate local information about the use of registers and stack slots at every machine instruction, plus information on the structure of prologue and epilogue, arguments, and return value. The complete output of the customised compiler is then passed on to a postprocessing stage, which performs the liveness analysis described in Chapter 5. The resulting PC maps, in the form of assembler directives, are then assembled, together with the rest, and a single object file is created. The advantage of having a single file containing both the compiled code and the related data structures (the PC maps used for pointer discovery) is that it is possible to maintain a symbolic association between code and maps, using the symbol tables embedded in the object code, which simplifies the later stages of the compilation process.

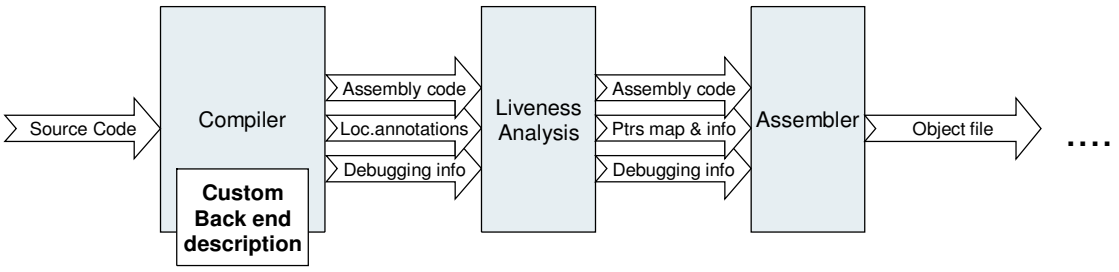


Figure 9.2.1: Compilation of one source file in the customised compiler

Once all of the object files have been created, it is possible to proceed with the creation of the executable file. The process is shown in Figure 9.2.2. The various object files are inspected, using the `nm` utility, in order to determine which routines have a PC map associated to them. A table is then built, associating the address range of each routine with the address of the PC map. Whenever execution is preemptively stopped, to run a system service, the master table is inspected to find out whether the current address is within a routine compiled with the customised compiler,

and if so where are the corresponding PC maps. Once that is done, pointer discovery can proceed.

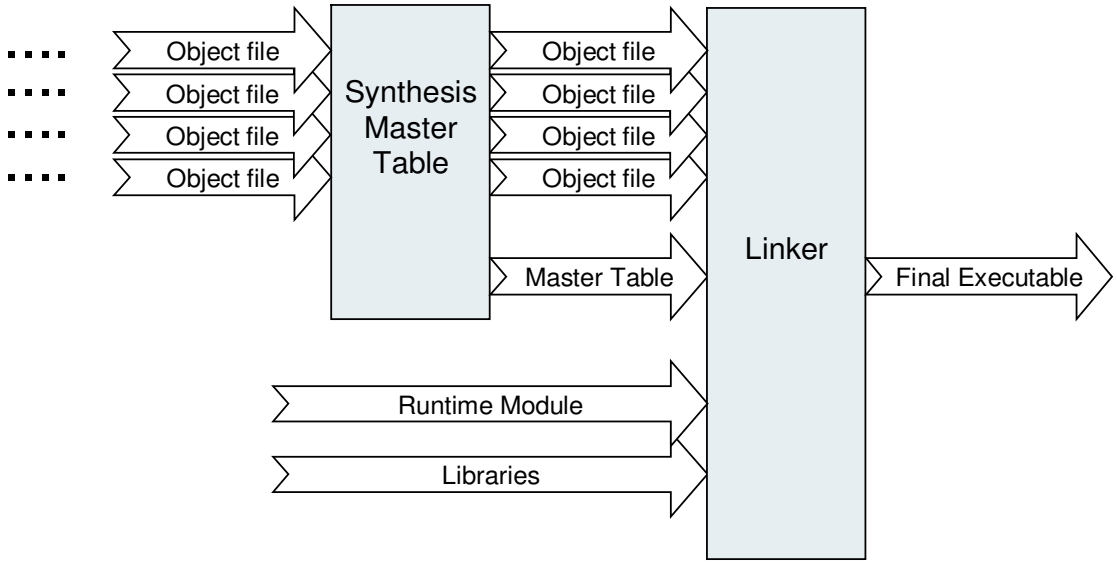


Figure 9.2.2: Generation of the final executable in the customised compiler

The master table is then linked together with the various object files, the runtime support for pointer discovery, and some general purpose libraries. The end result is a single file, that contains both the executable code and the additional infrastructure. In this implementation, some of the libraries and the runtime module are linked statically, while some additional libraries are loaded dynamically. At least in principle, it would be equally possible to link the runtime module dynamically as well, or even to load or change the program services at runtime, so that the very same executable can acquire different functionalities (garbage collection, persistence, and so on) depending on the running environment, or even change them on-the-fly.

The current runtime module has an overall structure very similar to the one shown in Figure 8.2.1 (in Section 8.2). The precise details of the runtime implementation, and the various components described above, will be now described together with the specific techniques used by each. To begin with, the following section offers some insight into the actual implementation of the customised GCC back end, and in particular the extraction of the local mode information for each assembly code instruction.

9.3 Extracting the mode information

Extracting the mode information for registers, and stack slots, used by individual assembly instructions, requires an inspection of the internal structures used by the compiler while generating the final code. While designing a new compiler, it should be fairly straightforward to add the necessary infrastructure. However, dealing with an existing compiler can be, in general, a much more complex task. First of all, it is necessary to make sure that pointer information is preserved

accurately in all the relevant cases, and modifications should be made where that is not the case. The resulting assembly code could be generated without following formal methodologies, and reconstructing the association between the uses of registers and the internal structures of the compiler, for instance, might be difficult. Finally, modifying the compiler code in order to emit the required information might involve altering substantial portions of the existing code, requiring considerable work and possibly introducing maintainability problems.

In the case of GCC, pointers are internally manipulated using the special machine mode "Pmode" (Section 9.1.1), and most of the infrastructure necessary to distinguish pointers from scalar values is already present. The segregation of pointers is rather strict, except in some special cases listed in Section 9.8. The way in which the infrastructure offered by Pmode can be used to distinguish pointers from scalars is discussed in Section 9.3.1.

In order to discover pointers at the level of detail of the single instruction, an additional liveness analysis might be necessary, as explained in Section 4.1. However, reconstructing the required liveness information for registers and stack locations during the compilation process might be rather difficult. The code generation in GCC, as seen in Section 9.1.2, is driven by a number of rewriting rules, optimisation descriptions, and other architecture-dependent definitions. All of those components are contained in a small number of architecture description files, which control the back end for a specific architecture. The final stages of the code generation are controlled by expansion rules, such as the "`movesi`" rule in Figure 9.1.2, on page 129, which can generate an arbitrarily long sequence of assembly instructions for each part of the RTL representation. The expansion is often represented by a piece of text, which is written in the output assembly code file, using pattern substitutions for some operands. As an alternative, the generated code can be the result of the execution of small fragments of C code, which are embedded in the expansion rules. Reconstructing on-the-fly the full life of registers and stack slots across those expansions, which can be more or less arbitrary, during the code generation stage would be rather difficult. In order to keep the complexity of the task under control, therefore, the liveness computation analysis has been delegated to an external postprocessing stage, while all that is required from the customised compiler is the detection of the mode of registers and stack slots locally to each assembly code instruction. The computation of the liveness information follows the model extensively discussed in Chapter 5. Before describing the way in which the mode information for each assembly code instruction was extracted, it will be useful to explain the the way in which pointers can be distinguished from scalars using GCC.

9.3.1 Partial integers

A crucial aspect of the mode extraction, naturally, is the ability to distinguish pointers from scalar values. In its internal representations of user programs, GCC associates to each expression a "machine mode", which indicates which low-level representation that expression will use [Staa]. For instance, a register can be used in mode "SI" (Single Integer) to refer to the manipulation of a 32-bit integer quantity, or the same register could be used in mode "QI" (Quarter Integer) if only 8 bits of that registers are used as an integer. Several different modes are supported by GCC, ranging from bitfields to large integers, floating points in various precisions, memory

blocks, condition codes and even complex numbers. The compiler will try to use the available characteristics of the current target machine, rearranging the code if certain modes or operations are not supported natively. The available features of each specific machine are described using some machine definition files, which form the machine-dependent part of the back end.

The ability of GCC to support many different microprocessors implies that the same compiler must be able to deal with all the peculiarities and special requirements that some microprocessor architectures may have. In particular, a little known, and little used, feature of GCC is its support for "partial integers", which is sometimes used to support those machines which have particular requirements for accessing their addressing space. For instance, while having 32 bit registers, certain machines could only be able to use 24 bits for memory addresses. Some machines could use a 16-bit register to address an 18-bit wide memory space, considering two additional lower bits as always zero. When the transformation between the numerical value of pointers and scalars is non-trivial, GCC allows the use of "partial integers" to indicate that only a restricted number of bits of an integer value can be used to represent a pointer. In those rare cases, GCC supports the necessary conversions between partial and conventional integers, and uses a separate set of rules to manipulate partial integers.

For all of the most common architectures, however, the GCC back ends simply manipulate pointers as normal integer values. All the rules and code generation schemes contained in the back end only ever refer to integer values and the distinction between pointers and non-pointers is not present. The mechanism used in this prototype was the reintroduction of the distinction between pointers and scalars in the back end, using the mode PSI (Partial Single Integer) to represent 32-bit pointers and the mode SI (Single Integers) for 32-bit scalars, even if such a distinction is not ordinarily present in the SPARC back end. The entire back end, in consequence of this modification, had to be thoroughly reviewed and adapted, since the possible RTL intermediate expressions produced during the code production and optimisation were now sometimes, but not always, referring to PSI values rather than SI, depending on the context. In many cases, RTL rewriting rules had to be substantially rewritten, or reproduced in multiple variants, according to the mode of the operands.

The distinction between integers and pointers, after the modifications, was enforced by explicit conversion patterns that GCC allows the back end to define. By controlling the operation of those patterns, it was also possible to keep under control conversions from scalars to pointers and vice versa or, where applicable, embed additional code in the output file to perform operations related to the conversion. The distinction is enforced everywhere by GCC except in certain cases, as explained in Section 9.8. In general, nonetheless, the distinction between PSI and SI values reflects with a very high degree of fidelity the distinction between pointers and scalars, and that distinction was used while extracting the mode information for the individual assembler instructions, as explained in the next subsection.

9.3.2 Customised expansions

In order to extract the local mode information, the code expansions were modified in order to generate, alongside the assembly code, additional annotations. In the previous "`movesi`" expansion

pattern (in Figure 9.1.2), the instruction that is generated to move a 32-bit value from memory to a register is `ld %1,%0`. The `%0` and `%1` are substituted, during the code generation, with the actual operands used during the expansion, so that the proper assembly instruction is produced. Other formats are available as well, and it is also possible to create custom ones. The back end was therefore modified, adding new formats, which are able to inspect the internal structures of GCC, using the standard interface between GCC core and back end. For example, the original expansion for the SI move can be transformed into:

```
(define_insn "*movsi_insn"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=r,f,r,r,r,f,m,m,d")
        (match_operand:SI 1 "input_operand"   "rI,!f,K,J,m,!m,rJ,!f,J"))]
  "(register_operand (operands[0], SImode)
    || reg_or_0_operand (operands[1], SImode))"
  "@
   mov\\t%1, %0\\n### %z1%Z0
   fmovs\\t%1, %0\\n### %z1%Z0
   sethi\\t%%hi(%a1), %0\\n### %z1%Z0
   clr\\t%0\\n### %Z0
   ld\\t%1, %0\\n### %z1%Z0
   ld\\t%1, %0\\n### %z1%Z0
   st\\t%r1, %0\\n### %z1%Z0
   st\\t%1, %0\\n### %z1%Z0
   fzeros\\t%0\\n### %Z0"
  [(set_attr "type" "move,fpmove,move,move,load,fpload,store,fpstore,fpmove")
   (set_attr "length" "1")])
```

The custom formats `%z` and `%Z` are used to print in the output file annotations about the use of the operand at that point. The formats `%z` is used when an operand is read and the format `%Z` when an operand is modified. The handlers for those formats, in the customised back end, automatically inspect their operand, extract the mode and print the relevant annotations. For instance, if the operand is a memory access, the expressions involved in the address calculations are recursively inspected and the appropriate annotations generated. The resulting annotations are printed on a special comment line, which follows the instruction to which it refers, with the following meaning:

- **>** a register is read as a pointer
- **^** a register is written as a pointer
- **:** a register is read as a scalar
- **#** a register is written as a scalar

For example, in the simple C program:

```
long mov(long *p) { return *p; }
```

the body of the function is expanded, if no optimisation is applied, into:

```
st %i0, [%fp+68]
ld [%fp+68], %g2
ld [%g2], %g2
mov %g2, %i0
```

With the additional annotations, the result becomes:

```
st %i0, [%fp+68]
### >%i0
ld [%fp+68], %g2
### ^%g2
ld [%g2], %g2
### >%g2 #%g2
mov %g2, %i0
### :%g2 #%i0
```

In the example, no additional annotation is printed for `%fp` since we know that the register always contains a pointer, during the entire program execution.

The architecture-dependent definitions, contained in the back end description files, that refer to prologue and epilogue generation have been similarly customised, so that the annotations regarding arguments, return values, and saving and restoring of registers are properly generated. The following section explains in some more detail the way in which registers are used in the standard SPARC ABI, and the information that is necessary to save in the SPARC v8 ABI in order to discover pointer usage in prologue and epilogue.

9.4 Pointer discovery in the registers

9.4.1 Registers in the SPARC ABI

According to the standard ABI for the SPARC [Spa92, Spa94], the general-purpose registers visible at any time to the programmer are divided into four groups of eight registers each: input, output, local, and global. Those registers are just a subset of the real, larger bank of physical registers contained in the microprocessor. When a new routine is called, the "window" of visible registers is changed so that the names used by the user program to refer to registers refer to different registers. That allows the program to achieve the same effect as saving some registers while new, unused registers are made available. In the SPARC, the output registers (used as outgoing arguments) are renamed and become available to the callee as input registers, while two new banks of local and output registers becomes available. Global registers are unaffected by the register renaming. The callee freely uses the new local and output registers, and fetches its arguments from what are now the input registers. The final results are left in the same registers and, when the routine ends, a renaming in the opposite direction takes place: local and output registers are discarded, and the input registers are again visible as output registers. The previously hidden local and input registers of the caller are made visible again. The results are therefore available, at the end, in what the caller sees as the output registers.

Some registers are reserved: the first global register, %g0, is always zero; the last output register, %o7, is used to save the return address during calls. Output register %o6 is the stack pointer (also named %sp), while input register %i6 is used as frame pointer (also named %fp). This particular choice causes the frame pointer and stack pointer to change role each time the register window is

shifted, so that the stack pointer of the caller coincides with the frame pointer of the callee. Also, the address of the call instruction, which is %o7 at the beginning of the prologue, is accessible as %i7 in the body, between the two window shift operations. No local register is reserved for special purposes.[1]

Every routine finds its first six parameters, after the window shift, in %i0..%i5, and the remainder (if any) on the stack. All six registers are assumed to be volatile across routine calls. Register %o7, which usually receives the return address during a call, is also used as volatile temporary storage, in between calls. For what concerns global registers, the manual of the SPARC V8 (appendix D) says: "The convention used by the SPARC Application Binary Interface (ABI) is that %g1 is assumed to be volatile across procedure calls, %g2..%g4 are reserved for use by the application program (for example, as global register variables), and %g5..%g7 are assumed to be nonvolatile and reserved for (as-yet-undefined) use by the execution environment." GCC uses %g1..%g4 as volatile temporary values[2] and never touches %g5..%g7. Floating point registers, in the SPARC V8, are used as temporary values or result values (oddly, not to pass arguments), exclusively to store floating point values. They can be therefore safely ignored as far as our mode analysis is concerned.

It should be noted that the SPARC V9 architecture, in 64-bit mode, uses registers differently. In particular, the SPARC 64-bit ABI allows the user code to use registers %g2/%g3 as application registers, while %g6/%g7 are used as OS registers. Also, in certain circumstances, floating point registers are used as general purpose registers, which includes parameter passing (so a floating point register *can* contain a pointer). This preliminary implementation, however, refers exclusively to the SPARC V8 32-bit ABI [San90].

Leaf routines (routines that do not call any other routine) usually do not perform a window shift, but directly use output registers (containing the parameters) and the global registers as scratch registers. A window shift can however be performed if some stack space is necessary (the window shift instruction is also used to allocate stack space) or if there would be too few free registers available without performing the window shift.

Summarising:

- For optimised leaf routines no stack space is allocated. Registers whose modes are only defined by the current routine are %o0..%o5, %o7 and %g1..%g4, while registers %l0..%l7 and %i0..%i5 have their mode defined by the caller.

- Non-optimised leaf routines and non-leaf routines will locally define %l0..%l7, %o0..%o5, %o7. Registers %g1..%g4 and %i0..%i5 (the parameters) are assumed to be volatile across calls and are therefore defined locally as well, be they parameters or local values. Consequently, no registers have their mode defined by the caller while the body is being executed. As will be shortly discussed, in the prologue before the window shift and in the epilogue

[1] Local registers %l1 and %l2 are used by the hardware when traps (including interrupts) take place. However, the registers are only used after a window shift is automatically performed and are therefore new registers, private to the trap handler. The local registers of the interrupted routine are really unaffected.

[2] If lexically nested routines are used, register %g2 stores a pointer to the static call chain, in conjunction with the "trampoline" mechanism described in Section 6.12. In that case %g2 does not contain pointers to heap objects, and can be ignored with regard to pointer discovery.

after the window shift the modes of registers %l0..%l7, %i0..%i5 are defined by the caller, with the same name they had there. Registers %o0..%o5 contain parameters/return values, and will become %i0..%i5 inside the body. Register %o7 is the return address (and as such it is reserved).

Finding the mode of registers is therefore particularly easy, since all the information is completely local most of the time. Whenever there are globally defined registers, it suffices to check the tables corresponding to %i7 (the caller's return address) to find the modes of %l0..%l7, %i0..%i5, without any need to perform register renaming. The remaining registers have either fixed use (%i6, %i7, %o6, %g0, %g5..%g7) or their mode can be determined locally, either because they are parameters/return values or because they are used as temporary values.[3]

9.4.2 Prologue and epilogue

Particular care must be adopted while dealing with the prologue and epilogue. It is necessary to determine, for each instruction, which registers have been saved, and for which ones the mode can be determined by considering the local tables. In the case of the SPARC V8, the situation is quite simple. In the standard implementation of prologue and epilogue generation, in recent versions of GCC, no register which is used to store temporary values can ever contain pointers to heap blocks. Additionally, the renaming takes place in a single step, using the standard instructions `SAVE` and `RESTORE`. Between the occurrences of the two instructions (`SAVE` in the prologue and `RESTORE` in the epilogue) there is no visible register whose mode depends on the caller, because of the previous discussion. In leaf functions, or in non-leaf functions outside the `SAVE`/`RESTORE` pair, the mode of registers %l0..%l7, %i0..%i5 can be discovered by looking at the tables referring to %i7 (the return address). The only other registers that can contain pointers to data blocks are those used as incoming arguments or for the return value. Within a `SAVE`/`RESTORE` pair, %i0..%i5 are the registers that can be used as incoming arguments, and %i0 as the return value. They assume the names %o0..%o5 and %o0, respectively, outside the `SAVE`/`RESTORE` pair. The mode of those arguments and return value also represent the initial and final modes of registers in the routine body, and as such are used in the implementation of the liveness analysis described in Chapter 5. Summarising, for the SPARC V8 architecture, the only data structures necessary for prologue and epilogue are: a flag that tells us whether `SAVE`/`RESTORE` are actually used in this routine, and the offsets in the code of the `SAVE` and `RESTORE` instructions. The only remaining information necessary to perform the pointer discovery in the registers can be obtained from the data structures related to the routine body, specifically the modes of registers throughout the body, and the modes of incoming arguments and return value.

[3] This analysis refers to the use of register windows detailed in the SPARC V8 ABI. GCC also supports an alternative "flat" model which does not use register windows, saving instead the registers on the stack. This less common model, which is however compatible with the general calling conventions, is more closely related to the traditional stack-saving technique used in other microprocessors, and was not considered in this implementation.

9.5 Implementation of the liveness analysis

The assembly code file, containing the code and the additional annotations, is postprocessed by a custom analyser, which implements a liveness algorithm similar to the one described in Chapter 5. The local annotations available for each assembly instruction, the information obtained about the arguments, and the information available on the return value are combined in order to reconstruct the mode of the various registers (and of the stack slots) throughout the routine body. In detail, the postprocessor first scans the input file and separates the assembly instructions from the annotations. The assembly code is copied, unchanged, to the output file, while the annotations are stored in memory for later processing. During a second pass, the delay slot information is analysed, and the internal representation of the code is rearranged by following the delay slot elimination algorithm described in Section 5.4. As a following step, the custom multi-mode liveness analysis is performed on the data. Finally, a number of sanity checks are performed and, if everything is in order, the liveness information can be written to the output file.

The liveness algorithm reconstructs the full mode information (whether a register is unused, used as a scalar or used as a pointer), but the runtime module of this implementation makes use only of the pointer information. The relevant part of the internal data structures describing the liveness is therefore extracted, compressed in order to save space, and written to the output file. The written data includes a header containing information on the routine (leaf/non-leaf, length of compiled code, offset of `SAVE`/`RESTORE`, and so on) plus the main table representing locations of pointers for each machine instruction in the routine body.

9.5.1 An example

As an example of the results obtained by the customised liveness analysis, Figure 9.5.1 shows a test program, written in C. The code does not perform any real function, but was written just as an example of use of different routines which manipulate pointers and scalar values in several ways, have different arguments and return values of different modes. An easily readable representation of the resulting mode maps, automatically generated by a customised version of GCC 3.3.3, is shown in Figure 9.5.2.

Once the structures generated by the liveness algorithm are ready, it is necessary to encode them in order to write them to the output file in a reasonably compact form. First of all it should be noted that certain registers are reserved for specific system functions, and it would be consequently useless to store mode information for them. For the remaining registers, the simple approach is simply to use a bitmask for each value of the PC, in which, for example, a bit set to ‘1’ means pointer and ‘0’ means scalar or unused. However, simple compression techniques can be used to reduce dramatically the space required.

```
typedef struct type_a {
 int a;
 char c;
 int *p;
 int b;
 char *cp;
} type_a;

typedef struct type_b {
 unsigned int u1,u2;
 type_a *a1,*a2;
 int *ip;
 unsigned char cc;
 char *pc;
} type_b;

int ping(int *);
int *pang(int,int*);
int pong(int*,int);
void peng(int);
int *poing(int);

type_a *some_a();
type_b *some_b();
char   *some_char();
int    *some_int();

type_a *fun1(int,type_b *,char,int,int*);
void fun2(int,int,int,int,int,int,int);
type_b *fun3(int*,int,int*,type_a*,type_b*,char*);

void rather_complex_test(int a,int *b)
{
 char *c=some_char(),*d=some_char();
 int o1=a,o2=a-3,o3=a+o1;
 int *po1=some_int();
 type_a *ta1=some_a();
 type_b *ta2=some_b();
 o1=1;
 do
 {
  int i,j=0;
  for (i=ping(b);i<ping(poing(j));i++) {
   peng(a+ping(pang(a,b)));
   while (pong(poing(a),pong(b,1))) {
    if (j<91) {
     ta2->cc=1+(*(ta1->cp=some_char()));
     peng(a-1);
     o3=-19;
     j=pong(b,a+2) ?  a*ping(b) :  0;
    } else {
     j=ping(pang(j,b));
     break;
    }
   }
  }
  fun2(o1,o2,o3,o2,o1,ping(poing(o2)),143);
  while (pong(poing(a),pong(b,1))) {
   if (j<91) {
    peng(a-ta2->u2);
   } else {
    j=ping(pang(j,b));
    break;
   }
   ta1=fun1(o2-8,ta2,*c,*po1,b);
  }
 } while (some_b()!=fun3(some_int(),*po1+ping(pang(3,ta1->p)),
                         po1,ta1,some_b(),d));
}
```

Figure 9.5.1: Example code

The data contained in the table appears to be easily compressible. As seen in the example, several registers are loaded with a value and the value is retained for long stretches of code, in

which the register mode does not change. A simple run-length encoding can be therefore used to reduce considerably the space needed by those portions of the tables. In certain cases, some

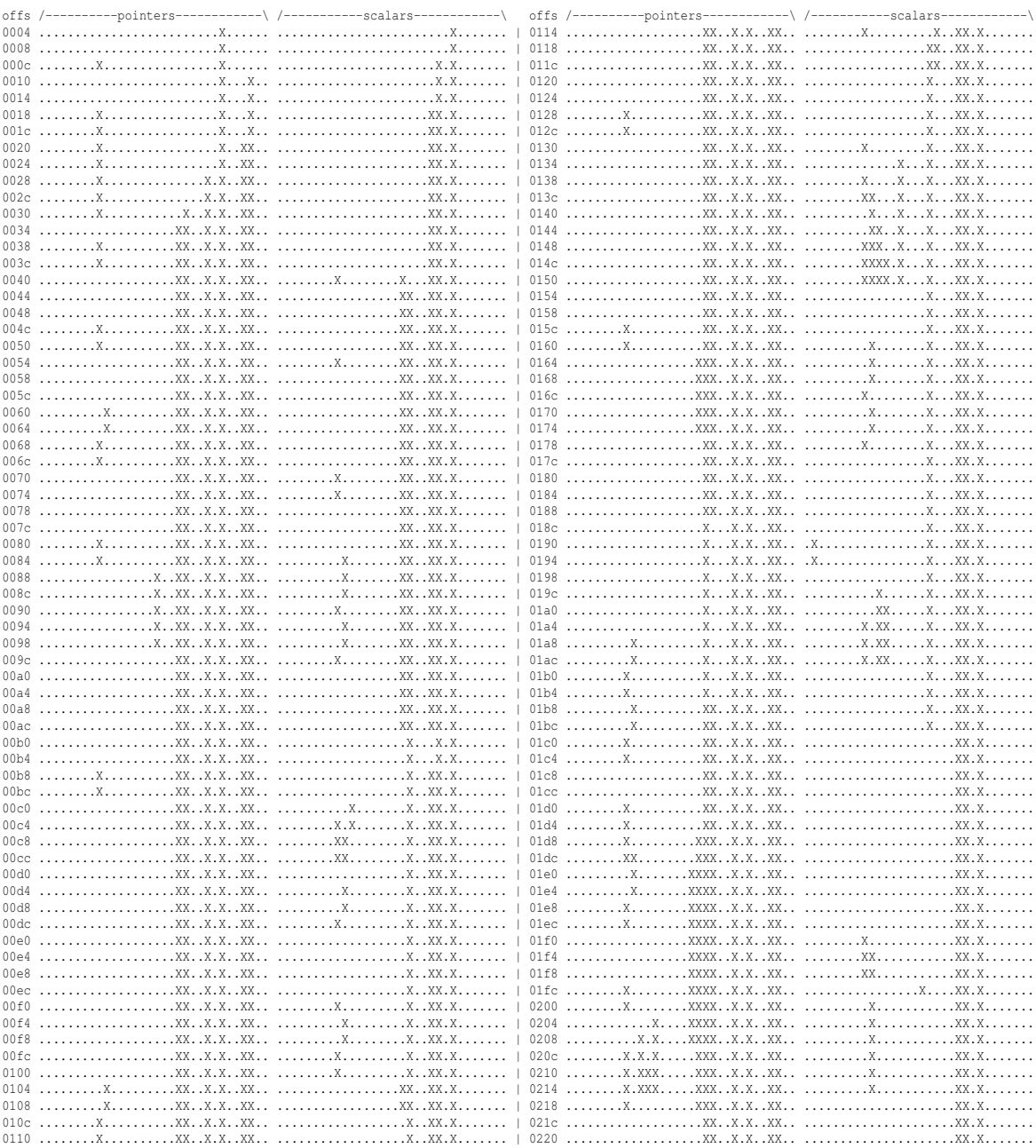


Figure 9.5.2: Tables showing the use of registers across compiled code. A mark is present in the column corresponding to each register if that register is used as a pointer or scalar, respectively, at that instruction.

registers are loaded with temporary values and used immediately. At times the same registers can be reused over and over again, possibly changing mode fairly frequently. This last pattern occurs, presumably, when the compiler picks the first available free register from a predefined list to hold temporary values. Since the registers available to the allocator are kept in a fixed ordered list, the first registers in the list tend to be used more frequently.

A suitable compression scheme might, therefore, exploit those properties to improve efficiency. A simple compression scheme has been devised to test the degree of compressibility of the tables. While the scheme has no pretence of optimality, the achieved results are rather interesting. A description of the technique follows.

9.5.2 A custom compression scheme

The compression scheme adopted is a variation on the classic run-length encoding, with some adaptations. The table that describes the use of pointers is compressed along the columns. If a column contains only ones, or only zeros, its content is not packed, but a flag is set in a separate word. The contents of the remaining columns are concatenated, and the resulting bitstring is compressed using the following patterns:

Seq	Len	Format	Uncompressed equivalent
A	4	0XXX	Verbatim: the 3 bits XXX as they are
B	10	10XXXXXXXX	Verbatim: the 8 bits XXX as they are
C	10	110NNNNNNN	Sequence of N+9 bits to ‘1’ (up to 136)
D	12	111NNNNNNNNN	Sequence of N+10 bits to ‘0’ (up to 521)

The scheme offers an easy way to deal with short bursts of rapidly changing states, while longer homogeneous sequences can be compressed using a short representation. Substrings of one, two or three ‘1’s can be encoded, together with some of the following bits, using sequence A, while longer irregular bursts can be encoded using sequence B. Up to 8 bits set to ‘1’ can be encoded using B, longer sequences are encoded using C. Up to 9 consecutive bits set to ‘0’ can be encoded using As and Bs, longer sequences using D.

The compression algorithm works by trying to locate substrings of at at least 10 bits set to ‘0’ and substrings of at least 9 bits set to ‘1’, which are compressed respectively using sequences C and D. All other substrings are encoded using As and Bs, so to obtain the shortest possible result. It can be shown that the best encoding (using A and B) for substrings n bits long (n>12) is a B plus the best encoding for the remaining n-12 bits, which makes finding the best sequence trivial.

If an A or a B precedes a C or D, some bits might need to be taken from the number of bits represented by the C or D in order to reach the minimal length of 3 or 8 bits respectively necessary to A or B. The compression can be implemented so that it runs in linear time with respect to the length of the input sequence. The final compressed bitstring is prefixed by a single additional header bit. If the header bit is zero, the remaining part of the bit string is interpreted as uncompressed data. That allows the encoded representation to be, in the worst case, one bit longer than the original table. If the compressed form happens to be longer than the original data, the uncompressed table is used instead (with one additional header bit).

This simple technique is designed to introduce only a minimal overhead for short bursts of bits set to ‘1’ and ‘0’ in what would be, if no pointers were ever used, a single stream of ‘0’ bits. At the same time, longer strings of bits set to ‘1’, useful to represent registers containing pointers that survive for some time, can be encoded using fairly short sequences. The scheme and the compression technique probably do not achieve the best possible compression level, but are just an attempt at a dedicated form of compression suitable for this particular application.

Nonetheless, in spite of the simplicity of the technique, the results are quite interesting, reducing considerably the space occupation of the packed tables. Although no extensive testing has been done, the memory necessary for the register discovery tables, excluding a short fixed-size header, was usually less than one quarter of the corresponding SPARC V8 code size. For example, the long example of Figure 9.5.2, compressed with this technique, is transformed in the short compressed data block shown in Figure 9.5.3, on the next page, which also shows in its entirety the full information that the runtime module uses to perform pointer discovery in this test implementation.

```
__TT__rather_complex_test_regTable:
###                     --Save and restore offsets for register window shift
###                       or add/sub offsets for stack pointer adjustment
        .long   0x00000000
        .long   0x0000022c
###                     --Start of body and Start of epilogue offsets
        .long   0x00000004
        .long   0x00000224
###                     --Flags: saveRestoreUsed
###                              retIsPtr
###                              spMoved
        .long   0xa0000000
###                     --Frame size:
        .long   0x00000078
###                     --Outgoing params area size:
        .long   0x00000068
###                     --Number of stack slots ever used as ptrs:
        .long   0x00000000
###                     --Used offsets:
###                     --Registers only used as scalars
        .long   0xff0706b3
###                     --Registers only used as pointers
        .long   0x00000040
###                     --Table columns contain:
### Reg/offs: %o0,%o1,%o2,%o3,%o4,%l0,%l1,%l2,%l3,%l4,%l7,%i4,%i5
###                     --Header completed.
###
###                     --Compressed table:
        .long   0xc4f7b460
        .long   0x607c04dc
        .long   0x10c07c02
        .long   0xdc11665e
        .long   0x70cbf07b
        .long   0x70e370e5
        .long   0x9903f433
        .long   0xa73d373c
        .long   0x3b70cbb7
        .long   0x67605ca8
        .long   0xedc1b847
        .long   0x026e7c03
        .long   0xa7700e24
        .long   0x001bb403
        .long   0xbb86f800
###                     --Table done.
```

Figure 9.5.3: Compressed tracking map

9.6 Discovery in stack and heap

9.6.1 Pointer discovery for the stack

Determining whether a certain stack slot in the current frame contains a pointer or not, at a given point in the code, can be fairly easy using GCC. During the various compilation passes that manipulate RTL expressions, all local variables and temporary values are handled by the compiler using an unlimited number of virtual pseudo-registers. During the final stages, GCC tries to assign those pseudo-registers to hardware registers and stack locations depending on the current level of optimisation chosen. If a high level of code optimisation is selected, GCC will try to use hardware

registers whenever possible, using stack locations otherwise.

Consequently, the treatment of both hardware registers and stack locations representing local variables is actually rather similar. The SPARC, in particular, has a limited set of addressing modes and it is relatively easy to generate annotations for stack slot access in a manner similar to that used for registers. Each access of a single local variable or temporary value on the stack is coded by GCC as an indirect access to a base register (either the frame pointer or the stack pointer) plus a constant. The liveness analysis performed for registers was therefore extended to include this case as well, using frame offsets alongside the real user-accessible registers. The internal components of the customised back end, which inspect expressions in order to generate annotations, were modified in order to generate annotations concerning the use of frame locations. A single table, containing the liveness information for both those frame locations and the microprocessor's registers is finally generated by the postprocessor. At this stage in the development of the prototype, it has been assumed that automatic variables are only accessed by the routine that allocates them. The cases, described in Section 6.7, in which a variable can be accessed by different routines were not implemented, and are currently unsupported.

The simplified liveness analysis, used in the current prototype, also does not include the case of arrays allocated on the stack. Section 6.7.5 discusses in detail the issues related to supporting stack-allocated arrays. While implementing pointer discovery in stack-allocated arrays is likely to be relatively easy, the feature was not included in this initial prototype. More details about the use of arrays are available in Chapter 10, "Derived Pointers."

9.6.2 Heap implementation

According to the general description made in Chapter 7, a simple custom heap manager was designed for this application. Although there are no particular reasons why an existing system should not be used or adapted, in this particular case it made sense to design a small heap manager with just a minimal set of features, in order to simplify the development and the debugging of the remaining, much more complex parts of the system. At present, memory blocks are allocated by specifying, at runtime, the layout of the block that will be created on the heap, as detailed in Section 7.1.1, and in particular where the pointers are situated in each block.

The simple heap manager does not perform any form of garbage collection, although that could be easily added on top of the existing infrastructure. However, the manager has the ability to compact the allocated memory, packing all the allocated memory blocks towards the bottom or the top of the heap. Each time a compaction is performed, all the pointers to the moved blocks present in the registers and the stack are properly updated, and the cross-pointers present in the heap are updated as well. The memory area previously occupied by the allocated blocks is filled with zeroes, to make sure that, when the program is resumed, the image of the memory blocks in use is really the one in the new location, and that the pointers used by the resumed program have really been updated to reflect the new memory configuration. The described implementation was successfully used in tests where linked data structures were created in the heap, and relocated transparently on-the-fly multiple times. After each interruption, served preemptively, the tests resumed their operation using the new configurations of the heap, and the updated pointers

contained in registers, stack, and heap.

9.7 Runtime module implementation

The runtime module, from the point of view of the developer, is one of the most complex components of the system, mainly because of the fine technical details related to the proper handling of all the characteristics of microprocessor and operating system. Parts of the runtime system, for example, are executed within interrupt handlers, and may need to modify system information saved on the stack, or even to add or remove frames, preserving the integrity of the stack structure, as expected by the operating system. In certain cases, the code that needs to be executed has critical restrictions on the way in which registers can be used, or some system information can be lost.

The code of the runtime module, in this implementation, was written mainly in C, using some GNU extensions to control the details of the generated code, and some portions of embedded assembly code in the most critical code fragments. The system was implemented on Sun Microsystems SPARC machines, using the v8 ABI, using different versions of the Solaris operating system. The current implementation of the runtime module, as a consequence, depends to a certain extent on the specific information extracted by Solaris when a signal is sent to a thread. Adapting the implementation to a different Unix-like operating system on the SPARC, however, should be a mostly painless task. The intimate connection of the implementation of the runtime with the structure and functionality of the SPARC, however, implies that a substantial rewrite of many crucial sections would be needed in order to support different microprocessors.

When a preemptive request for a service routine is received, the first part of the runtime module that assumes control is the signal handler. The signal handler has at its disposal the context of the thread that was interrupted. After some bookkeeping, the value of the program counter, in the suspended thread, is checked against the master table and the tables for pointer discovery are located. If a table is found, control is passed to the pointer discovery code, and then to the service routine, otherwise a few intermediate steps are required.

The runtime module implements the deferring technique mentioned in Section 8.5, offering support to single-threaded applications. If the running thread is interrupted in a portion of code that has no associated maps, one of the return addresses present on the stack is modified, so that it points to a deferred execution routine, and control is returned. When the microprocessor encounters the modified return address, the deferred pointer discovery takes control, and the service routine executed. In order to avoid multiple requests being deferred, a lock is held while a pointer discovery is pending. At the end of the service routine, the running context is restored, and control is returned to the original code. More details appear later in Section 9.7.1.

The stage of pointer discovery, therefore, can either take place immediately, or be deferred until the thread re-enters the code compiled with the customised compiler.[4] In the runtime im-

[4] In principle, only certain operations that can be done from within a signal handler, since many operations might result in a corruption of system data structures. If complex operations, for instance I/O, are required during the service routine, performing those operations from within the signal handler would be problematic. A possible approach would be to defer the execution of the service routine, in any case, until after the completion of the signal handler execution.

plementation there is also a provision for invoking the pointer discovery and the service routine synchronously, if so desired, directly from the user program with an explicit call. When the main pointer discovery component assumes control, the stack is scanned downwards, one frame at a time, and the locations containing pointers in each frame are added to a set of locations. It may be useful to recall that the registers in use in the user thread at the moment of the interruption are, at this stage, saved somewhere in main memory. The addresses of the locations used to store those registers, in case they contain pointers, are added to the previous set as well. Once stack and registers have been processed, the heap is scanned as well. The locations containing pointers in each heap block are added to the previous set, and the service routine can then perform the necessary heap manipulations.

The service routine has access to the set of all the memory locations that may contain pointers to heap blocks. In certain cases, as explained in the previous chapters, some of those locations might contain values that appear to be legal pointers to heap objects, but are actually unused. In that case, however, it is still possible to alter freely those values, since no live scalar value will ever be contained in those locations. Another important aspect is that, because of the information that the compiler maintains internally, we are only able to discriminate pointers from non-pointers, but we are unable to distinguish statically among different kinds of pointers (to heap, to stack, to code, and so on). It will be necessary, therefore, to check dynamically the values contained in those locations to distinguish possible pointers to heap blocks from the remaining pointers. In an ideal case, the compiler should be able to maintain such a distinction statically, whenever possible, which would reduce the work at runtime. Notably, some programming languages only allow pointers to the heap, or require a specific syntax to differentiate pointers to heap blocks from generic pointers, but this distinction is not preserved down to the final code by usual compilers. (Consider, for instance, Ada aliased access types).

Once the service routine has completed its job, control can be returned to the user program, which will resume its execution. Some implementation aspects of deferring the service routine, and of the pointer discovery, will now be discussed.

9.7.1 Deferring the service routine

The actual code that implements the deferred pointer discovery is quite complex, having to deal with rather subtle architectural details of the SPARC. A more detailed discussion on the techniques used might be interesting to have a better understanding of the implementation issues involved. In particular, it will be shown how the presence of register windows and delay slots, while a source of considerable complication, are no obstacle to the functionality of the system.

For instance, it is possible to create “manually” an additional stack frame in which the return address is the former value of the PC, and all the registers have been saved on the stack, while the PC has been replaced with the address of the pointer discovery routine. The pointer discovery, and the service routine, can then operate at the end of the signal handler, and before execution of the user code is resumed. A simplified approach, used in this implementation, assumes that, in a single-threaded program, if the value of the program counter refers to code compiled with the customised compiler, then at that particular moment no system code is running, so the system data structures are in a consistent state. Consequently, it should also be safe to execute operations that are not normally safe for execution during a signal handler. The previous, more complex approach should be used, in theory, to fully respect the letter of the manuals.

When a signal is received, the signal handler sets a lock that will be held while a service routine is pending. A pointer to the `ucontext` structure, containing crucial information about registers and stack in the interrupted thread, is passed to the handler. In the case of the SPARC, the `ucontext` contains some of the user-accessible registers, while the remaining ones are left in a register save area on the stack. In detail, the `ucontext` stores the flags register PSR (Program Status Registers), the program counter PC, the register nPC (whose value is to be loaded into the PC after the execution of the current instruction), the temporary register %y (used as a scratch register in certain operations), and the registers %g0..%g7 and %o0..%o7. The register %o6 is also the stack pointer, and there is a way from that value to find the most recent register save area on the stack, containing the remaining registers %l0..%l7 and %i0..%i7.

At this point, it is necessary to verify whether the saved value of the program counter corresponds to a position in the code in which it is possible to reconstruct the location of all the pointers to heap blocks. Superficially, it appears sufficient to verify that the PC refers to a portion of code generated by the customised compiler. In practice, however, there might be cases in which code generated by the customised compiler calls foreign code, which in turn calls code generated by the customised compiler, for instance by means of a callback handler, or a pointers to a function. In that case, even if the PC points within "safe" code, there could be frames, allocated on the stack by the foreign code, containing pointers to heap blocks, and it is not possible to discover exactly where those pointers are.

The situation is the one represented in Figure 9.7.1. In order to make sure that all the pointers can be discovered, therefore, it is necessary to wait for the user thread to complete all of the pending foreign code, so that the frames left on the top of the stack belong to code compiled by the customised compiler. That is equivalent to finding a subchain of the dynamic call chain that goes from the address within the first used routine in the customised code, deep into the stack, up to the first address in the foreign code, if present. Every return address in that subchain must refer to a routine in the code generated by the customised compiler.

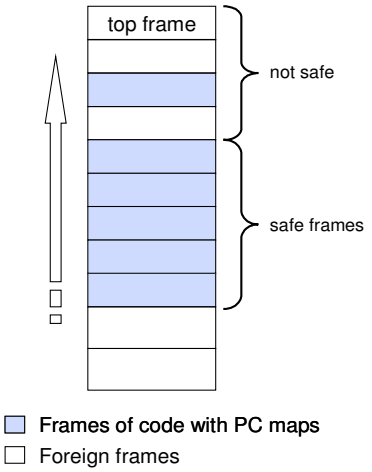


Figure 9.7.1: Stack containing mixed stack frames

On the SPARC, because of the register windows, the situation is particularly complex. The call instruction automatically saves the most recent return address in the register %o7. If a register window shift is then performed by the called routine, that register becomes known as %i7, while a new %o7 is ready to accept a new return address for a further subroutine. That implies that the dynamic chain sometimes includes %o7, and at other times it does not. The register %o7 will contain the most recent return address if execution is stopped in a leaf routine, or in a non-leaf routine before the `SAVE` instruction, or after the `RESTORE`. If %o7 is used, the following return address is in %i7, and the rest are on the stack. On the other hand, if execution is stopped between the `SAVE` and the `RESTORE` instructions in a non-leaf routine, it is %i7 that contains the

most recent return address, and the rest are on the stack.

This peculiar situation is complicated by the fact that, in general, there is no way to tell whether execution was stopped between a `SAVE`/`RESTORE` pair just by looking at registers and stack. It may happen that a source-level debugger on the SPARC gets confused about the stack layout, especially when stepping through individual machine instructions. In our case, however, the additional data structures associated with each compiled routine can be used to determine whether `SAVE` and `RESTORE` are used, and their positions with respect to the value of the program counter. Nothing is known, however, about the routines of the foreign code. Determining the correct subchain of the dynamic chain, as previously discussed, is rather tricky.

As a preliminary step, we consider just %i7 and the rest of the stack, and we search for the longest subchain with certain characteristics. We need a subchain that is entirely made up of addresses with PC maps associated to them, and such that all of the addresses in the stack, below that subchain, refer to foreign code. There are three possible cases:

1. A subchain was found, and it includes %i7. It means that the most recent return addresses, excluding possibly PC and %o7, all refer to "safe" code, generated by the customised compiler.

2. A subchain was found, but it does not include %i7. Some frames, on the top of the stack, will have to be discarded before the service routine can operate normally.

3. No subchain was found. This might happen if the signal handler has just been installed, but execution of the user code has not begun, yet. Alternatively, it might mean that we were stopped while executing the main routine, or one of the routines it calls, therefore the contents of PC and %o7 must be checked.

It is now possible to determine which return address that must be patched, in order to cause the deferred execution of the service routine. The full scheme, with all the alternatives, is as follows.

- If a subchain not including %i7 was found, then the return address to patch is the topmost of the found subchain. That address will be considered when returning to our code from foreign code. At that point, it will be safe to execute the pointer discovery and the service routine.
- Consider the content of PC. If the PC refer to code with maps associated to it:
  - If no subchain was found, it means that either PC or (PC,%o7) are the topmost addresses that we are interested in. In both cases, it is safe to proceed with the service routine immediately.
  - If a subchain was found, including %i7, there are two subcases, depending on whether %o7 is currently used or not. Since the PC refers to code generated by the customised compiler, it is possible to look up the tables and to find out whether %o7 is really used.
    - If %o7 is not used: it is safe to proceed with the service routine immediately
    - If %o7 is used: is it in foreign code?

      · If the address in %o7 is in foreign code: it is necessary to wait for that code to complete. The return address contained in %i7 is patched (saving the previous value) and execution is deferred.
      · If the address in %o7 is not in foreign code: it is possible to proceed immediately.

- The PC refers to foreign code. How was the subchain previously found?
  - No subchain was found. The only possible address that could refer to a routine in the customised code is in %o7, but the register could be not in use so it is not enough to rely on the value it contains. However, if %o7 really refers to "safe" code, it has to refer to the main routine in the user program. In that case %i7, the previous address in the dynamic chain, must forcibly point to the (only) calling site in the runtime module itself that calls that routine. Conversely, if %i7 has any other value then %o7 cannot refer to that main routine. Checking %i7 against that address, therefore, allows us to tell whether %o7 refers to the main or not.
    * If %o7 refers to the main in the user program: patch %o7 (saving the previous value) and execution is deferred.
    * If %o7 does not refer to the main in the user program, then there is no "safe" code currently in execution, and the request for the service routine should be dropped.
  - A subchain, including %i7, was found. The current value of PC, however, refers to foreign code so execution must be deferred. Register %o7 could be in use or not. The value can be checked using the master table.
    * If the value of %o7 is not in the range of addresses of the code generated by the customised compiler, then %o7 is either unused or it refers to foreign code. In either case, %i7 can be patched (after saving its value) and execution of the service routine is deferred.
    * If %o7 appears to be a valid pointer in code for which discovery would be possible, it is not possible to tell for certain, in this particular case, whether %o7 is used or not, and therefore whether %i7 or %o7 should be patched. In this case, it is never completely safe to patch %o7 (it could be used as a regular scratch register in a leaf routine), and %i7 should be patched instead, even if that means having to wait for a further routine to complete, in the worst case. A further heuristic is actually used in the implementation, relying on details like the opcodes of call instructions and similar details, to guess whether %o7 is actually in use or not.

Due to the peculiar characteristics of the SPARC architecture, occasionally it may happen that altering %o7 or %i7 is not enough to cause the return instruction to jump to the deferred handler. For instance, consider the following code fragment, which might be found at the end of a compiled routine:

```
jmpl [%o7 + 8], %g0 ; retl
st %o2, [%o4 + 4]   ; some instruction in the delay slot
```

The return instruction is actually a jump instruction, with which the address contained in %o7 is reloaded into the PC. Because of the delay slots, however, PC is not changed immediately. Instead, the value in %o7 is loaded into the additional register nPC, and the microprocessor executes the instruction in the delay slot. After that, nPC is loaded into the program counter, and the control flow continues with the caller. If an interrupt occurs exactly before the execution of the instruction in the delay slot, modifying %o7 is not sufficient, because the target address has already been transferred in nPC. As a consequence, whenever %o7 or %i7 are patched, it is also necessary to check whether nPC is equal to the old value of the register which is being patched, plus the fixed offset. If that is the case, nPC must be changed as well.

After all the work above is done, either the service routine is executed immediately or one return address has been patched. In the latter case, the runtime module simply returns from the interrupt and waits for the control flow to be intercepted again. When that happens, the patched address, corresponding to a deferred handler, is followed and the handler assumes control. That handler appears, in a way, to be called "during" a return instruction, therefore great care must be taken in order not to corrupt any value contained in the registers. In the initial part of the handler, the registers declared as volatile across calls, excluding %o0 which may contain a return value, are available, and are used while creating a new context, in which the remaining registers are saved. The pointer discovery can then proceed normally. At the end of the service routine the registers are reloaded as they were before control was intercepted, and the normal control flow is resumed.

9.7.2 Pointer discovery implementation

The implementation of the pointer discovery is also rather complex, involving an analysis of the different cases in which the user program might be stopped. Essentially, the process of pointer discovery is subdivided into four distinct phases. The first phase is determining the pointers in the registers whose modes are determined by the current routine. During a second phase, the register save areas in the stack are scanned. At each step, the modes of the registers contained in the reserved area of each frame are detected. During the third phase, the automatic variables in the various stack frames are analysed, and the pointers they contain are discovered. In the final phase, the heap is explored, and pointers found. The various phases will now be briefly examined.

9.7.2.1 Registers

In order to determine which registers have their modes defined by the local routine, it is necessary to determine whether execution was stopped in a leaf routine or not, and exactly in which part of the compiled routine. For non-leaf routines, these are the possible cases:

1. Prologue instructions before `SAVE`
2. The `SAVE` instruction

3. Prologue instructions after `SAVE`
4. Body instructions
5. Epilogue instructions before `RESTORE`
6. The `RESTORE` instruction
7. Epilogue instructions after `RESTORE`

Without going into excessive detail, these are the conclusions for the different cases:

Case 4: The modes of all registers are available locally. To be precise, registers %l0..%l7, %o0..%o5,%o7, %g1..%g4 and %i0..%i5 can contain pointers, and their mode can be checked in the maps for the current routine.

Cases 1&2: The modes of all registers depend on the caller. In principle the maps corresponding to the return address should be used to determine the modes of all registers. However, those maps are not useful to determine the modes of the registers used as incoming arguments. The modes corresponding to the first instruction of this routine can be used for that purpose. Therefore:

- %l0..%l7,%i0..%i5 can be determined from the maps related to the return address
- %o0..%o5 can be determined from the mask bits for %i0..%i5 for the first instruction of the body

Case 3: The modes of all registers is available locally. In particular, the modes will not be changed until the microprocessor reaches case 4. Therefore the modes for the same registers listed for case 4 can be obtained from the maps relative to the first instruction in the body.

Cases 5&6: All registers are defined locally, but the only register that can be used at this point is %i0, used as a return value. Its mode can be obtained from the tables for the current routine.

Case 7: All registers depend on the caller, similarly to cases 1 and 2. Therefore:

- %l0..%l7,%i0..%i5 can be determined from the maps related to the return address
- %o0 can be determined using the same information used for %i0 in cases 5 and 6.

In cases 1, 2, and 7, in which it is necessary to inspect the maps corresponding to the program counter in the caller, the return address can be obtained from %o7+8 (the same register becomes %i7 in cases 3, 4, 5, and 6). The reason for the "+8" is that %o7 contains the address of the call instruction, but the return address actually corresponds to the instruction following the instruction in the delay slot, therefore two instructions beyond the call, each four bytes wide. In cases 1, 2, and 7, the address of the most recent stack frame (which is the one used by the caller) is contained in %o6, while between `SAVE` and `RESTORE` the address of the frame of the caller is contained in %i6.

In the case of leaf routines, %l0..%l7,%i0..%i5 are always determined by the caller, and are never touched throughout the present routine The return address is always contained in %o7, for leaf routines. There are just three possible cases:

1. Prologue instructions
2. Body instructions
3. Epilogue instructions

Case 1: The modes of %o0..%o5, which are used as incoming arguments, can be obtained from the maps corresponding to the first instruction of the body.

Case 2: Registers %o0..%o5, %o7 and %g1..%g4 are defined locally, and their mode can be obtained from the maps for the current routine.

Case 3: The mode of %o0, used as return value, can be obtained from the local tables.

9.7.2.2 Register save areas

After the registers have been examined, the pointer discovery routine explores the register save areas of the routine whose completion is pending. The analysis is performed by scanning down the stack, one frame at a time. For each frame, an analysis similar to the one above is made, and the maps for the corresponding routine are used to determine the modes of the registers saved on the stack. In principle, the return address should refer to a location within the routine body, but it might also refer to the first instruction immediately following the body. The return address is, of course, not the instruction directly following the call instruction but the one following the instruction in the delay slot. Knowing that each register save area stores registers %l0..%l7 and %i0..%i7, the following applies:

Case 1: If the return address is contained within the body, then the local maps can be used to establish the modes of the saved copies of %l0..%l7, %i0..%i5.

Case 2: If the return address is after the body, then only %i0 is alive, containing the return value from the called routine.

After scanning each frame, the content of %i6, as saved in the stack, can be used to find the following frame, until the scan is fully scanned.

9.7.2.3 Automatic variables and stack-based arguments

Obtaining the modes of the automatic variables present on the stack, in each frame, can be done by traversing the stack and exploring the maps at each step. While this phase can be easily combined with the previous one, in the current implementation the two are kept distinct for clarity. Together with the stack-based automatic variables, this phase determines the modes of the stack-based incoming arguments, if any. In the SPARC v8 ABI, only the first six arguments can be passed in reserved registers, while the rest are stored on the stack.

Each automatic variable, or argument, is identified using the offset of the memory location from a base register, which can be the stack pointer or the frame pointer depending on whether `SAVE`/`RESTORE` are used or not. In leaf routines, where no window shift is applied and therefore the frame pointer cannot be used, the stack pointer is used instead. The offset of the variable, however, is relative to the address contained in the base register within the routine body.[5] During the prologue or the epilogue, the base register could still have the same value used by the caller. The first step, therefore, is to determine, using the value of the program counter saved in the context, whether the base register for that routine has been changed or not. There are two possible cases.

If `SAVE`/`RESTORE` are used, then the frame pointer is used as a base register between the two instructions. If the user program is interrupted before the `SAVE` in the prologue, or after the `RESTORE` in the epilogue, the frame pointer has not been moved yet to reserve the necessary space on the stack. In that case, the offsets will have to be adjusted accordingly, so that the true locations can be determined. In detail, during the prologue there are no automatic variables in use on the stack. In that case, the stack-based arguments will be the only elements to consider. During the epilogue, conversely, there are no automatic variables in use, and the arguments are no longer alive, therefore it is not necessary to look for pointers.

If `SAVE`/`RESTORE` are not used, the leaf routine might still create some space on the stack by adjusting the stack pointer manually. The standard routines that generate prologue and epilogue, in the back end, have been patched so that an indication of such adjustment is passed on to the final discovery tables. It is still possible to check the program counter, therefore, against those values and find out whether the stack pointer was moved to reserve space on the stack or not. In this case as well, if the user program is stopped in the prologue there might be pointers in the frame slots used for the arguments, while no stack slots are used during the epilogue.

After the first frame has been inspected, the previous values of the stack pointer and the return addresses are successively extracted from the register save area, and the same procedure is repeated for the following frames, until the stack is completely scanned. Similarly to what happened in Section 9.7.2.1, the return addresses might point just after the last instruction in the body, and in that case the frame content is ignored, since there are no frame locations reserved for automatic variables or arguments that can be in use at that moment.

[5] If `alloca()` or similar features are not used (see Sections 6.7.5 and 6.8), there is a fixed distance between the frame pointer and the stack pointer, in the body of non-leaf routines. All offsets, for simplicity, are actually adjusted by the prototype to use the stack pointer as reference register, even when the frame pointer is used instead in the actual code.

9.7.2.4 Heap

Once the registers and the stack have been analysed during the previous phases, the heap is scanned using the descriptors specified during the allocation of each heap block. The automatic generation of layout descriptors, as described in Section 7.1.1, has not been implemented in the current prototype. The descriptors are therefore created manually, using the ABI specifications as a guide to the layout structure.

While, in principle, the heap scanning could be decoupled from the service routine itself, in this implementation it is the service routine that decides which heap blocks need to be scanned, and does so by calling the runtime module. The reason is that not all of the blocks present in the heap might need to be scanned. If a reachability-based analysis is performed, for instance, the pointers present in registers and stack act as reachability roots. Only the heap blocks that are transitively reachable from those roots will need to have their pointers scanned, and possibly adjusted, while there is no need even to check the structure of the unreachable blocks.

9.7.3 Service routine

The service routine can preemptively inspect and manipulate the custom heap using the information supplied by the runtime module. The main purpose of this prototype was to expose the less obvious problems that might arise when implementing a support for preemptive system services. The objective, therefore, was to build a system able to obtain all of the information required to run preemptively a service routine, and in particular able to discover all the pointers in the heap, in the stack, and in the microprocessor registers. The specific service routine to use was not particularly important, being a test rather than an essential component of the system.

In this implementation, the test routine which was used simulates operations similar to those which might be done by a compacting garbage collector. Each time it is called, the test service routine relocates all the heap blocks, compacting them alternately towards the beginning and the end of the heap memory area. After moving all the blocks, the area of memory previously in use is filled with zeroes, so that any attempt to dereference a pointer which was not correctly updated will most likely result in a program crash, or at least in erratic behaviour. In the tests conducted so far, all the pointers in use appeared to be discovered and updated as expected.

9.8 Arrays and GCC

In certain cases, in particular when dealing with arrays, GCC may fail to maintain internally a strict separation between scalars and pointers. The problem stems from the structure of the predefined RTL patterns which must be defined in the back end to describe basic operations like sum, difference, and so on. All those standard patterns are required by GCC to operate on arguments of the same mode. For example, there is a definition of the sum of 16-bit integers, obtaining a 16-bit integer as a result, a different definition for the sum of 8-bit integers, and so on.

The same mechanism, however, also applies for partial integers. From the point of view of GCC, partial integers are just a numeric representation of legal values for pointers, handled as

integer with some restrictions on the number of usable bits. The way in which partial integers are used, on the other hand, is as a completely separate representation of pointers, as explained in Section 9.3.1. A sum of partial integers therefore assumes the connotation of a sum of pointers, which has no practical use, nor much meaning.

The primitive operations on partial integers are not normally used directly by GCC, since there is no practical reason why a compiler would want to sum or multiply two pointers. Conversely, we would possibly be interested in a sum of a pointer and a scalar value, or a difference between pointers obtaining a scalar, in the style of C. In most cases, the way in which the code is expanded can be controlled by adapting as needed the various rewriting rules in the back end. In one case, however, the expansion is performed autonomously by the GCC core, and it is not possible to alter this behaviour without patching some additional code in the GCC core.

In detail, GCC might attempt to perform arithmetic on partial integers while generating the code corresponding to array accesses. In that case, a base register (pointer) and an index (scalar) must be combined to obtain the address of an individual array element. GCC expands the resulting expression by converting the offset from integer to partial integer, and then attempting to perform a homogeneous sum between partial integers. From a purely numerical point of view, that operation works perfectly well. However, performing such a conversion between a scalar and a pointer is illegal for us, since we want to enforce a strict separation between pointers and scalars. The resulting RTL expressions, and the corresponding generated code, would handle the offset as a partial integer, which means that a scalar value (the offset) could be mistakenly identified by the runtime module as a potential pointer, and possibly updated as a result of memory movement.

There are at least two possible solutions to this conundrum. The first, fairly easy to implement, involves a modification of the postprocessor, and of the custom liveness analysis, in order to track backwards the origin of pointers. If a pointer is discovered to be the result of a sum, the two addends (which look like pointers) can be treated as potential scalars. Tracking recursively the origin of each of the subexpressions, there should be no problem in eventually reaching the leaves of the expression, which are either pointers or scalars. At this point it is trivial to proceed in the opposite direction, resolving the uncertainties and determining exactly which values are indeed true pointers and which are not. This additional analysis can be seen as a sort of workaround on the particular way of generating array access expressions by GCC.

A possible alternative, cleaner but more complex, would be to alter the portions of GCC that expand the array access expression, forcing it to use patterns which sum integers with partial integers, if available. In that way the annotations generated while the assembly code is created would be more accurate, and there would be no need to modify the postprocessor. The downside of this method is that the GCC core itself has to be modified, which introduces a number of problems involving compatibility with the original back ends, portability on newer versions of the compiler, and general maintenance issues due to the possible internal dependencies on the code that is being modified. Conversely, altering the postprocessor allows all of the modifications to be confined to the back end, whose interface with the GCC core is much better defined and relatively stable. The full implementation of either of the two alternatives, in the context of a more complete support for derived pointers as described in this chapter, was left as a future development. The implementation of the modified postprocessor, in particular, seems to be quite straightforward.

A further case in which a scalar may be converted implicitly to a pointer is the use of null pointers. If a pointer variable is assigned "null", the corresponding low-level operation is equivalent to moving a constant (zero) into the pointer variable, which implies such a conversion. In practice, however, the assignment is easily handled by the back end with a specialised rewriting rule that recognises the special case, and does not perform any mode conversion. The assignment is eventually transformed to a "clear pointer" operation, which poses no problems.

9.9 Limitations/Future developments

This test implementation was essentially a test case, designed as a challenge to show the difficulties in developing such a system. For that reason, a choice was made of the set of features to be supported, so that a working system, properly interfaced with GCC, could be obtained within the time allotted. Nonetheless, the current implementation can certainly be expanded and improved in a number of areas, offering a more realistic support to real-life programming environments.

None of the features that were left out during this first stage of development present extraordinary implementation difficulties, and they have all been discussed in the previous chapters. With respect to the C language, it should be pointed out that some features of the language cannot be easily supported by such a system, due to the possible loss of pointer information. In particular:

- Casting a pointer to a scalar value (int or long) can cause the compiler to lose track of the pointer as such. That could cause a reachability analysis not to identify all active pointers and memory blocks.
- The standard C memory allocation routines (`malloc()` and related operations) does not require the programmer to identify the structure of the allocated blocks. The resulting insufficient information, available to the heap handler, would not be sufficient to identify the active pointers present in the heap. For that reason, the standard allocation routines must be replaced with custom calls.

Other features of C and other languages could, in principle, be supported according to what is discussed in this thesis. The main features not current supported by the present prototype, and left for future development are the following:

- Packed records (like ANSI C bitfields) can cause pointers to be aligned to arbitrary bit offsets, requiring bit-level precision in the various tables. Currently pointers are required to be aligned to a 4-byte boundary.
- Multithreading, which requires the runtime system to be specially designed in order to support multiple contexts and stacks.
- The additional support necessary for the treatment of exceptions is missing.
- Derived pointers can only be handled using special techniques. They are discussed extensively in the next chapter. Currently only pointers to the base of heap blocks are supported. That involves the following limitations:

  - Large records, which cause the microprocessor to perform intermediate address calculations, are not supported
  - References to inner fields of records cannot be extracted
  - Support for arrays is currently incomplete

- Jump tables (used, for instance, in “switch” statements) imply multiple possible target destinations for jumps. Simple branches are supported in the current implementation, but not jumps with multiple destinations.
- Arbitrary unions would require a reorganisation of the fields in the records, which is not performed.
- Global are currently not inspected. The related implementation is likely to be straightforward, since the structure of the global area is normally fixed for the entire life of the program.

9.10 Testing

The prototype was tested both verifying statically the consistency of the generated tables and data structures with the compiled assembly code, and running dedicated test programs in order to verify the ability of the system to manipulate memory preemptively.

9.10.1 Static testing

A core part of the prototype is the automatic generation of PC maps, relying on the information available to GCC, on the customised back end, and on the mode analysis. In order to veryfy the ability of the prototype to generate PC maps, a number of test programs were created and processed with the system. The limitations in the supported features, as detailed in the previous section, did not allow us to experiment with standard benchmarks or pre-existing programs. Consequently, the test programs were especially designed and tailored to test the system behaviour.

In particular, test programs were used to test unusual situations, like routines with an unusually large number of parameters, or (using automatically generated test code) particularly long routines, or code containing expressions so pathologically complex that the set of available registers, even on the SPARC, is not sufficient to store all of the intermediate results, forcing the spilling of some temporary values on the stack. Test programs written in multiple languages were also used, and the system was able to successfully generate PC maps from code written in C, C++, Ada, Pascal, and Java. The resulting tables were painstakingly manually inspected in order to check their consistency with the corresponding compiled code. The compiled code itself was also checked against the code generated by the unmodified compiler, in order to pinpoint discrepancies and irregularities. A number of built-in consistency checks (as detailed in Section 5.3) were also used to verify the internal consistency of the tables.

The tables encode, with a short header, certain caracteristics of the routine as, for instance, the offsets of the beginning and the end of the body, the use of `SAVE/RESTORE` in prologue and epilogue, and so on. The PC map, possibly compressed, follows the header. Since the exact life of scalar values is not essential to relocate memory, only pointer information is preserved in the final PC maps. Consequently, their size can vary considerably, in accordance with the greater or lower use of pointers in the test programs. In one especially crafted example, which used almost exclusively pointers, the maximum size of the compressed table reached at most one third of the code size, but the results for every other test, using more balanced code, was well below 25% of the code size. Some examples, with the related statistics, are available in Appendix C.4. The compression routines themselves (following the technique described in Section 9.5.2) were exhaustively checked using automatically generated test patterns, compressed and decompressed to verify the correctness of the implementation.

9.10.2 Dynamic testing and debugging

Due to the limitations of the current prototype, it was possible to perform only a limited testing of running programs. The main aim of the testing performed was to verify the full functionality of the current prototype, checking the consistency of the internal structures and verifying that running code, using the custom memory manager, can indeed be suspended preemptively and resumed flawlessly after the heap content has been modified on-the-fly.

The dynamic behaviour of the system depends on the runtime module, which, upon receiving an interrupt, takes care of scanning the PC maps, reconstructs the pointer information, passes control to the service routine, and finally returns control to the user program after performing the necessary adjustments. Especially tricky, as explained in Section 9.7, was testing the reconstruction of the stack layout and the actions used to support deferred interrupts. Initially, explicit synchronous calls were inserted in the compiled calls to verify the ability to parse the stack, and find the pointers in stack and registers relying on a stable and known frame configuration. Subsequently, the interrupt handler was connected to a console signal, and signals were manually sent to the running program. All the possible cases of contents of PC, %i7 and %o7, as described in Section 9.7.1, were checked exahustively. A great help during this stage was offered by the Simics simulator suite, developed by Virtutech, which allowed the code execution to be inspected step-by-step, regardless of the system state. Execution was therefore traced through interrupt handlers and stack frames reorganisation. The interface available between Simics and the GDB debugger was also invaluable. Notably, part of the debugging was actually performed simulating a complete SPARC Solaris system on a conventional PC running Linux on x86 (albeit at a fraction of the speed).

In order to test some running code, an example which generates and deallocates multiple binary trees in C was created, and its execution repeatedly interrupted sending signals asynchronously, both manually and from a concurrent deamon program. Upon reception of the signal, the simple service routine implemented in the system moved all of the allocated memory blocks alternatively towards the top and the bottom of the heap memory, setting to zero the previously occupied areas. No actual collection was performed, since the main point of the test was verifying the ability to find all of the pointers, rather than perform the actual memory manipulation (dependent

on the specific service routine). Finally, the test code was expanded and rewritten in different languages, so that a single executable containing C, Pascal and Ada code was generated and run successfully with preemptive memory manipulation. Each of the portions in the different languages created and deleted binary trees, and a main code in C called each of the portions. Tests in which the data structures were created using one language and deleted using a different one were also successfully completed. A test involving Java at runtime was not readily feasible, due to the need to coordinate the Java system libraries with the customised heap handler. Nonetheless, PC maps for code fragments written in the Java language were successfully generated, within the limits detailed in Section 9.9.

9.11 Results

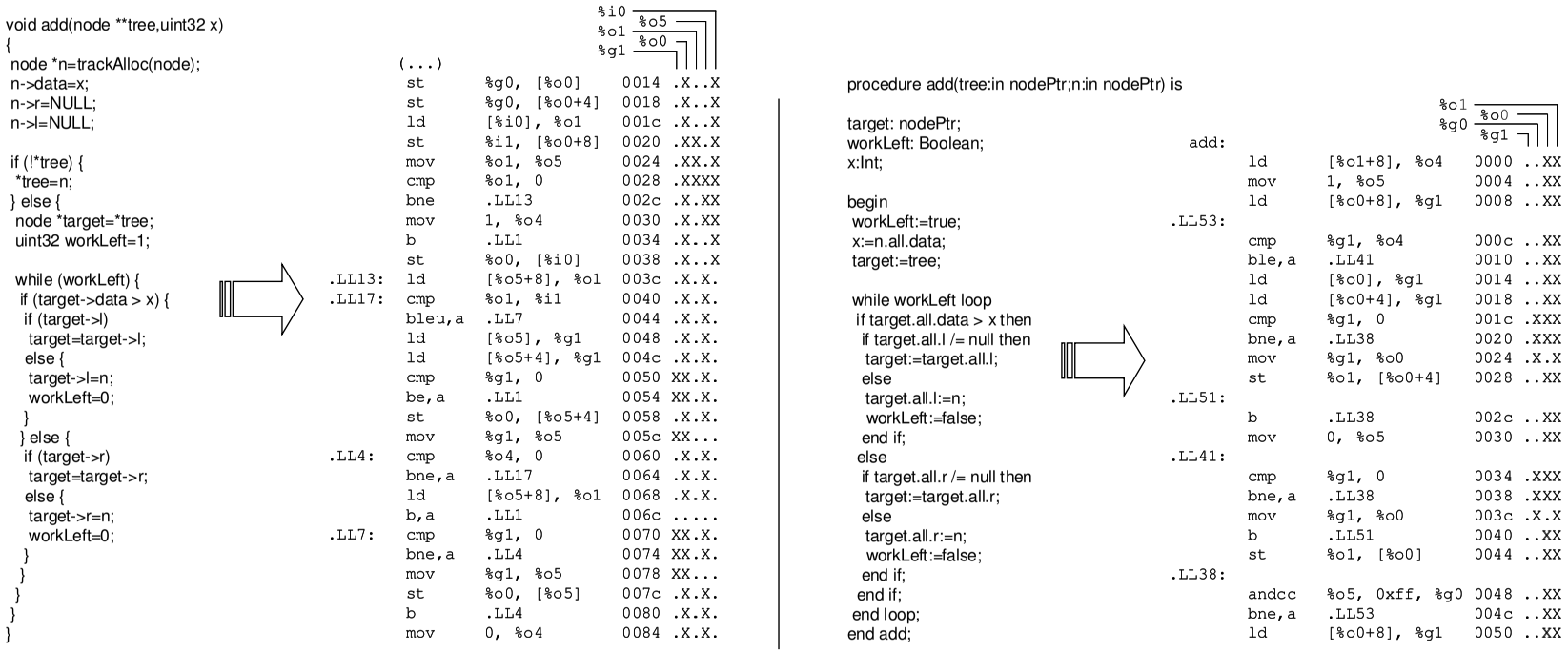
The various techniques and algorithms described in the previous sections were the basis for the creation of the test implementation. The SPARC back end of the GNU Compiler Collection (GCC) was modified in order to create the local annotations, and the necessary postprocessors and runtime components were developed. In particular, confining the modifications to the compiler back end allowed development to progress throughout multiple versions of GCC, from the initial version 2.95.2 to the most recent version 3.3.3 (released 14 February 2004), with relatively little effort.

The existing GCC front ends did not require customisation, allowing the prototype to support multiple high-level source languages, albeit with the limitations listed in the previous section. This is the first time in the literature that a system that uses this kind of maps can be used with multiple high-level languages. Test programs written in C, Pascal and Ada were successfully compiled, using the highest level of optimisation. The maps necessary to discover the pointers were successfully generated automatically, and the final executable ran normally. It was possible to send at arbitrary moments a signal to the running program, causing the intervention of the service routine. After the heap manipulation, the running program resumed operation, totally unaware of the alterations made to the pointers contained in its registers and memory.

It would have been interesting to compare the performance, and measure the possible advantages in terms of latency, of such a system with a similar system that, while using preemptive thread switching, relies on more conventional safe points. In the case of this prototype, however, there was no reasonable way to proceed with a realistic comparison, both because of the limitations of the current implementation and because the main focus of this work was investigating the implementation difficulties of such a system, rather than creating a production tool. Nonetheless, it is still possible to compare the code generated by the customised compiler with the code generated by the original GCC, to check whether the various modifications applied to the back end had a substantial effect on the efficiency of the generated code. In most cases, the results of the comparison show that the code generated by the customised compiler is absolutely identical to the code generated by the original compiler, or differs in trivial ways. For instance, the example shown in Section 9.5.1, on page 139, compiles to absolutely the same code in both cases, while every instruction in the generated optimised code is a safe point.

That shows that it may be possible to retrofit an existing compiler, in order to produce pointer discovery maps, even if the intermediate representation is not tied to a specific target architecture and even if the final code generation stage takes place without following formal transformations (and in GCC the code generation stage is most definitely not done according to formal models). That is in stark contrast to the approach followed by Stichnoth et al. [SLC99], in which the intermediate representation was designed to support a specific target architecture, to the point that each instruction in the intermediate representation matches exactly one instruction in the final code. A more flexible approach, instead, can be of help while adapting existing compilation infrastructures. An example of side-by-side comparison of the code generated is available in Appendix C.

The size overhead of the maps for each routine body has been found to be typically less than 25% of the code size. Such result is consistent with the observations of Diwan [DMH92], Tarditi [Tar00], and Stichnoth [SLC99]. The deferred execution of service routines was tested extensively and worked as expected. The feature was used to interface the code generated using the customised compiler with existing libraries. A concrete example of the liveness information calculated by the system, showing the tables generated for C and Ada code, is available in Figure 9.11.1. More examples are available in Appendix C.



(a) Optimised C code

(b) Optimised Ada code

Figure 9.11.1: Part of the generated tables for C and Ada code examples.
An 'X' is present in the tables if the corresponding register contains a pointer before the execution of that instruction.

Don't Panic.
*— **Douglas Adams**, The Hitchhiker's Guide to the Galaxy*

Chapter 10

Derived Pointers

10.1 Pointers and heap blocks

An important aspect of the ability to manipulate the heap preemptively is the ability to move heap blocks on-the-fly, while adjusting the value of all the pointers that refer to that memory block. That allows the system to implement features like compaction, or relocation while migrating memory (as opposed to isomigration [ABNP01]) while preserving the integrity of the data structures present in the system. As we shall see in this chapter, however, reconstructing the connection between the heap blocks and the pointers to them is not always straightforward.

The basic idea is that, if a heap block is relocated in the addressing space, the corresponding pointers need to be adjusted by the distance through which the heap block was moved. So far we have implicitly assumed that the association between pointers and heap blocks can be easily reconstructed. For instance, in the case of "base" pointers, which point to the base location of a heap block, the connection is obvious. In certain cases, however, the pointer might not refer directly to the beginning of the block, but rather to a different location. In certain cases the pointer might be pointing somewhere within the heap block, for instance a pointer to a field within a structure contained in the heap block. Such a pointer will be called "internal". If we know that a pointer is actually an internal pointer it is still possible to determine, although that might be computationally expensive, which is the block that corresponds to that pointer.

In other cases, however, the pointer might point externally to the heap block. Consider for instance the case in which an array is repeatedly accessed. It is a common optimisation technique, in that case, to pre-calculate the address of the "virtual origin" of the array, that is the address that an element would have if all its indexes in the array were zero. However, in the case in which the lower bound of one of the indexes is greater than zero, the virtual origin address might be lower than the address of the first real element of the array in question (and similarly for the higher bound). If the array is contained in a heap block, the virtual origin might refer to a location which is completely outside the block boundaries. If the pre-calculated address is stored in a register,

pointer discovery would still need, somehow, to reconstruct the connection between the pointer and the heap block, whether it is a base pointer, an internal pointer, or an external pointer.

More generally, some values might be the result of a computation that involves the address of one or more heap blocks. We will call those values “derived values”, to highlight the connection between heap block addresses and the calculated value. In particular, we will be interested in derived pointer. If some heap blocks are relocated, the derived values will have to be updated accordingly. In extreme cases, for instance in the C language, the computation that leads to those derived values (pointers or otherwise) can be completely arbitrary due to the ability to perform pointer arithmetic. Excluding those cases, however, the compiler itself will create derived values when performing particular optimisations, or while generating the code for well-known operations. The most common cases will now be summarised.

10.2 Common sources of derived values

The most frequent sources of derived values can be summarised as follows [DMH92, BC92]:

- Virtual origin for arrays, as previously discussed
- Common subexpressions, for instance `A[i,2]:=a; A[i,3]:=b;` might get converted into:

```
p=&[A,2];
*p=a;
*(p+1)=b;
```

- Strength reduction in loops, so that `for(i=0;i<10;A[i++]=a);` is compiled into:

```
p=&A[0];
for(i=0;i<10;*p++=a);
```

- Double indexing (rarely used). If two arrays are scanned in parallel, the offset could be calculated just once. For instance:

```
for(i=0;i<10;i++) {
 A[i]=a;
 B[i]=b;
}
```

  could be transformed into:

```
offs=(&B[0]-&A[0]);
for(i=0;i<10;p++) {
 *p=a;
 *(p+offs)=b;
}
```

- Parameter passing by variable, for instance passing a field in a structure as a parameter would create a reference to that field, or similar mechanisms by which a reference to a field or array element might be created implicitly by the compiler (the `WITH` statement, for example).

The cases described are the most frequent, and probably the most important to support in a real programming environment. In addition, split pointers, described in section 3.4.2, can also be seen as a kind of derived values. With the exception of the double indexing, and of expressions explicitly calculated by the user by means of pointer arithmetic, the remaining derived values are actually derived pointers which can only depend on the address of at most one memory block, if the structure or array to which they refer are contained in the heap. We will only discuss derived pointers, rather than arbitrary derived values. Being able to reconstruct at runtime the association between the derived pointer and the heap block which logically corresponds to it is the challenge.

Notably, the compiler described by Stichnoth et al. [SLC99] does not deal with derived pointers at all, keeping instead only base pointers in registers, presumably relying on the complex x86 addressing modes to perform array accesses and similar operations. Such an approach may be more problematic using a RISC machine, which has simpler addressing modes and therefore may require explicit address calculations to perform analogous operations. Additionally, avoiding the use of derived pointers prevents the compiler from applying the important optimisations listed above, reducing the efficiency of the generated code.

10.3 Dealing with derived pointers

10.3.1 Derivation tables

A possible approach to handling derived pointers is the use of "derivation tables", which describe, in a form accessible by the runtime module, the way in which the derived pointer was obtained by combining other values. The idea is that, knowing how a derived pointer resulted from a certain expression involving the base address of a heap block, it is often possible to perform an inverse calculation, reconstructing the base pointer from the derived pointer. Furthermore, once the heap block has been moved, the expression can be used again to recalculate the updated value of the derived pointer, which still refers to the same block.

The way in which derivation tables can be implemented is discussed by Diwan et al. [DMH92], in the context of a Modula-3 compiler based on GCC. The flexibility of derivation tables lies in their ability to support rather complex derivations, enabling the representation of derived values obtained in various ways. For instance split pointers, described in Section 3.4.2, could be seen as a kind of derived value, and the association between the split pointer and the complete pointer it refers to can be easily preserved in the derivation tables. On the other hand, the biggest drawback of derivation tables is the fact that the base values, from which the derived values should be recomputed if necessary, must be kept alive as long as the derived value can be accessed, even if those base values are no longer used by the code. That might cause inefficiencies in the compiled code, since the larger set of live values, at every instruction, could cause some registers not to be

reused as efficiently as in the fully optimised code. Other inefficiencies might arise from the need to determine statically the full derivation expression, which sometimes cannot be done if multiple control paths are merged together as a result of an optimisation. Another potential problem could be the additional memory occupancy for the storage of the tables, although that occupancy is related to the code size and does not scale up with the size of the data generated by the program, therefore it is unlikely to be a substantial concern. While derivation tables can be a very effective solution to handling derived pointers, other techniques may be worthy of some investigation.

10.3.2 A different approach

An alternative to derivation tables may be abandoning the attempt to preserve the exact association between the derived pointer and the base pointer of the object, but just preserving enough information to be able to reconstruct at runtime, indirectly, the association between the derived pointer and the heap block it refers to. The idea aims to associate a derived pointer, which may point anywhere, with a location which is certainly within the boundaries of the heap block to which the derived pointer logically refers. Knowing the internal pointer can then allow the heap manager to identify the heap block and determine the corresponding base pointer. There appears to be no previous mention of this technique in the literature. It is therefore presented as an initial proposal of the author of this thesis as an alternative to derivation tables, while further study may be useful on some aspects, as detailed at the end of this subsection.

The basic idea, as previously mentioned, is finding a way to determine an internal pointer from every derived pointer. In particular, we want to associate to each derived pointer, for every machine instruction, a fixed offset that, added at runtime to the current value of the derived pointer, gives us the address of a location within the corresponding heap block. That can be achieved by considering the ways in which, according to the list in Section 10.2, derived pointers can be generated by a compiler in the most common cases. If the derived pointer refers to a field of a structure, then the pointer is also an internal pointer, and it poses no problems. Conversely, accessing an array presents two kinds of derived pointers that can be used: pointers which are pre-calculated as virtual origins or as common subexpressions, and pointers which are used to access individual elements, for instance using autoincrements/decrements.

In the case of a pointer used to access an array element, unless the program behaves erratically, it can be reasonably expected that the pointer will actually be used, at runtime, to perform a read or write of a real element of the array or as a border condition, if the pointer is also used as an index, the pointer value might correspond to the first location after the end of the array. If the pointer is used to access a real element, then the pointer is actually an internal pointer. If it points after the end of the array, it can be still considered an internal pointer if we are able to differentiate between the memory block we are interested in and the following one in memory. For instance, if there is a header, containing at least one machine word, before heap blocks, a reference to the first address after the end of the block could be easily associated to the block that precedes rather than the block that follows.

The most complex cases are the calculation of a virtual origins, and the partial calculation of the address of a specific array element. In those cases, the address which is calculated could be

an external pointer, and there could be no fixed offset, that can be statically calculated, between the computed address and the beginning of the array. However, assuming that the element which will be eventually accessed exists, it should still be possible to determine a pointer which points somewhere within the array, although it cannot be determined statically where. The following discussion will clarify the proposed mechanism.

10.3.2.1 Array representation

In a language like Ada, for instance, we might have an n-dimensional array declared as `A[`$l_1..u_1, l_2..u_2, \ldots, l_n..u_n$`]`, where l_i and u_i are the lower and upper bounds of each dimension. We can assume, without loss of generality, that the array is stored in row-major order (as in Pascal, as opposed to the column-major order used by Fortran), that is an element `A[`$x_1, x_2, \ldots, x_n + 1$`]` is stored in memory immedately after an element `A[`$x_1, x_2, \ldots, x_n$`]`, an element `A[`$x_1, x_2, \ldots, x_{n-1} + 1, l_n$`]` is stored right after an element `A[`$x_1, x_2, \ldots, x_{n-1}, u_n$`]`, an element `A[`$x_1, \ldots, x_{n-2} + 1, l_{n-1}, l_n$`]` is stored right after an element `A[`$x_1, \ldots, x_{n-2}, u_{n-1}, u_n$`]`, and so on. Consequently, we can always treat an n-dimensional array as an m-dimensional array, with $m < n$, in which each element is a complete $(m-n)$-dimensional array.

What we would like to obtain is the ability to associate at any moment each pointer, which is the result of a virtual origin calculation, with the array to which it refers, so that the pointer can be properly adjusted whenever the corresponding array is moved. For instance, let us assume that a pointer referring to the virtual element `A[`$0, 0, \ldots, 0$`]` is calculated. If the bounds are known statically, or at least at the time when a memory manipulation is requested, we can easily establish that the physical representation of the array is separated from the mentioned virtual origin by the following number of bytes:

$$l_n + l_{n-1}(u_n - l_n) + l_{n-2}(u_{n-1} - l_{n-1})(u_n - l_n) + \ldots + l_1(u_2 - l_2)\cdots(u_{n-1} - l_{n-1})(u_n - l_n)$$

or, written in a simpler form:

$$\sum_{i=1}^{n} l_i \prod_{j=i+1}^{n} (u_j - l_j)$$

It is sufficient to add that number to the virtual origin to find the address of the physical array to which it refers, and therefore the corresponding memory block that contains the array. If some memory blocks are then moved, it is straightforward to adjust the virtual origin pointer, if necessary, to match the new location of the virtual element `A[`$0, 0, \ldots, 0$`]`.

10.3.2.2 More general virtual origins

The case of a pointer referring to `A[`$0, 0, \ldots, 0$`]` is actually quite specific, and other kinds of virtual origins can be calculated automatically by the compiler, and stored in temporary pointers. A simple case would be the sequence of operations that the compiler uses to calculate the pointer to `A[`$0, 0, \ldots, 0$`]`. It might happen that the compiler precalculates the constant, using the formula above, and then subtracts in a single step the constant from the initial address of the

array, in order to calculate the virtual origin. On the other hand, it might also happen that successive calculations are applied, obtaining a series of intermediate pointers. For instance, the compiler might first subtract l_n from the initial address of the array, obtaining the pointer to the virtual element `A[`$l_1,\ldots,l_{n-1},0$`]`. Then it might subtract $l_{n-1}(u_n - l_n)$, obtaining the address of `A[`$l_1,\ldots,l_{n-2},0,0$`]`, and so on. At every step, a temporary pointer might be stored somewhere, yet it is still necessary to reconstruct the association between such a pointer and the beginning of the array. Luckily, since the full calculation is under the full control of the compiler, it is straightforward to associate the correct displacement to each temporary pointer across the code that performs the calculation in the final compiled code. No particular problem exists in that case.

In other cases, however, the compiler might try to use optimisations that make it much harder to reassociate precisely the temporary pointer with the beginning of the array. Consider, for instance, the following code:

```
A[i,j,k]:=e;
A[i,m,n]:=f;
A[i,m,x]:=g;
```

where the various indexes are not known statically. Optimising the intermediate steps of the addresses calculations, the code could be optimised as:

```
t=&(A[i,0,0])
*(t+(j*(u2 − l2)+k))=e;
p=&(*(t+m*(u2 − l2)));
*(p+n)=f;
*(p+x)=g;
```

In this case multiple values, most of which are not known statically, are added to the base pointer. The only way to reconstruct the exact position of the beginning of the array from one of the intermediate pointers `t` and `p` would be to reconstruct the expressions using the values of `i`, `j`, `m`, and `k`, as would happen using a derivation table. As mentioned in Section 10.3.1, however, derivation tables can be complicated to represent, with a consequent memory occupation, and require all the base values of the expression to remain alive as long as any derived value is alive, which might potentially impact on the reuse of registers, and on the quality of the generated code.

A possible alternative could be, rather than looking exactly for the first location of the array, trying to find any location that is internal to the array itself. Such information is sufficient to reconstruct the association between the temporary pointer and the memory block. For instance, when the address of `t=&A[i,0,0]` is precalculated, the value of `i` is not known statically. Nonetheless, since a physical access is made to the element `A[i,j,k]`, we know that `i` is supposed to be, at that point in the code, a valid index and therefore within the range $l_1..u_1$. Regardless of the exact value of `i`, it is therefore known that the pointer referring to `A[i,`l_2,l_3`]` is a legal internal pointer within the array `A`. Additionally, we also know that the address of `A[i,0,0]` is exactly $l_3 + l_2(u_3 - l_2)$ bytes away from `A[i,`l_2,l_3`]`. Therefore, if we want to find a pointer which is internal to the array to which `A[i,0,0]` refers, we just have to add to it the value $l_3 + l_2(u_3 - l_2)$. What

has been done is, in a sense, considering the array as a monodimensional array of bidimensional arrays, and separating the two calculations.

Let us repeat the same procedure for the pointer `p`. The temporary value in this case is `A[i,m,0]`. Adding back to it the constant l_3, because of the previous reasoning, it is once again possible to obtain a pointer that, although its exact value is not known statically, is certainly going to be within the boundaries of the array itself. The core of the technique is that, regardless of the calculations performed during a virtual origin calculation, it is always possible to reconstruct from each of the temporary pointers an internal pointer. As long as there is at least one word separating arrays residing in distinct memory blocks, the technique is also suitable for pointers pointing to the first location immediately following an array, as allowed by the C language.[1]

This technique can avoid many of the problems of derivation tables. The amount of information that it is necessary to store is smaller, and typically the base values, from which the derived pointer is calculated, are not required to be alive during the whole life of the derived pointer, as it happens using derivation tables. Finding the heap block given an internal pointer, however, might be computationally expensive if the heap manager was not designed to support the operation efficiently, which might on the other hand require some overhead in terms of memory and/or allocation time. For that reason, the technique that is presented in this subsection presents some trade-offs in terms of the advantages, described earlier, versus the additional memory or speed penalty. It is important to point out, however, that finding which block corresponds to a certain internal pointer only needs to be done during the pointer discovery stage. That aspect could be exploited to minimise the overhead if the pointer discoveries are not frequent, reducing or eliminating the allocation speed overhead while increasing the (proportionally less frequent) cost of pointer discovery. Additionally, in many cases it should be possible to determine statically whether a given pointer is a base pointer or a derived one. That would allow us to further reduce the additional work, since base pointers can be dealt with in a simpler way. The various trade-offs should be carefully evaluated to determine the concrete viability of the solution in a given context.

[1] Notably, if indexes out of range by one position were allowed in languages with native multidimensional arrays, as in Pascal or Ada, the temporary pointer might reach beyond the location immediately following the array. For instance, in an array like `A[0..1,0..1]`, the virtual element `A[2,1]` would refer to a location two positions beyond the end of the array. That is not a problem in most languages, however: C arrays are monodimensional, while Pascal, Ada, and other languages do not allow out-of-range index values.

You can't have everything. Where would you put it?
*— **Steven Wright**, 1955 -*

Chapter 11

Pointer Discovery as an Enabling Technology

The use of the techniques discussed in the previous chapters can have a positive impact on certain aspects of computer systems. Being able to determine the complete set of pointers at an arbitrary position in the code, for instance, can be used to reduce latency when preemptive thread scheduling is in use. In certain circumstances, the ability of discovering pointers at any instructions may even be a requirement, a necessary step in order to implement certain memory handling techniques. A concrete example of the importance of pointer discovery in the development of new memory management techniques will be introduced in this chapter.

The particular topic, which will be described in some detail, was explored by the author during an internship in Sun Microsystems Laboratories, Mountain View, California. The interaction between the memory management technique and the main body of research of this Ph.D. will be discussed in Section 11.3.

11.1 Thread-local heaps: an introduction

The creation of an efficient parallel garbage collection algorithm presents a number of challenges, due to the presence of multiple active CPUs that need to be coordinated while performing their operations on the shared memory heap. For instance, multiple processors, while cooperating in the garbage collection, could try to create copies of the same heap block without being aware of one another, or they could be rearranging pointers in an uncoordinated way. The situation can be even more complex if, at the same time, some of the processors are performing garbage collection operations while others are concurrently executing normal user code, acting therefore as mutators on the same data structures which are being analysed by the collector.

In order to coordinate properly the work of the different microprocessors, some form of synchronization is necessary. The overhead involved can be sometimes significant, depending on the

specific details of the garbage collection algorithm and the overall organization of the memory heap. Many garbage collectors, for instance, are "stop the world" collectors, in which, when a garbage collection is required, all threads are stopped so that there are no active mutators in the system. Depending on the implementation, the overhead involved can be greater or smaller. For instance, an exact garbage collector, if no information is available about pointers in registers at all the points in the code, might need to roll forward, to the nearest safe point, all the threads in the system, which might involve a substantial delay.

While the synchronization overhead can be more limited using the most recent replicating garbage collectors [CB01, HM01], some degree of synchronization is still necessary, even in those state-of-the-art systems, to prevent multiple processors from interfering with one another while working on the garbage collection. Since every synchronization involves overhead, it may be useful to explore techniques used to reduce the number of necessary synchronizations.

11.1.1 Thread-local heaps

An interesting thread of research revolves around the partitioning of the common heap space in multiple areas using "thread-local heaps". [Ste00, Sal01, Har01, DKL+02, Kin03] In particular, each thread obtains a private, thread-local area, that contains only memory blocks that are guaranteed not to be accessible by other threads. A common, shared heap is also available to store the blocks that may be used by multiple threads. The potential advantages of such an organizations are quite interesting. Each thread becomes able to perform a garbage collection of its own local heap without the need of any synchronization or communication of any kind with other threads. During such operation there is no need to stop the other threads, which can continue undisturbed to act as mutators or can collect their own heaps, and each garbage collection does not need to perform any locking at all on the local heap or on the individual blocks. Furthermore, all memory allocations that are performed in the local heap do not need to be synchronized, hence the allocations in that space can be faster. A similar improvement can also be obtained using "local allocation heaps", in which a small portion of the heap is devoted to the allocation of blocks for each thread. However, in this case, no independent garbage collection is possible, since all heap blocks logically still exist in a single, common heap.[1]

11.1.2 The shared heap

While thread-local heaps allow the garbage collection to be performed without synchronization in many cases, the synchronization cannot be avoided for those heap blocks that really need to be accessed from multiple threads, for instance because they are part of a large shared data structure or because they are used for communication between different threads. A common, shared heap is used for this purpose, alongside the thread-local heaps. All the techniques that are normally used while handling a fully shared heap can still be applied in this case (for instance generational

[1] Sometimes, confusingly, local allocation heaps are also referred to using the term "thread-local heaps". In this thesis the latter denomination is used exclusively for logically distinct and independent heaps, to which a single thread has exclusive access. Another term used in literature for the same concept is "thread-specific heaps."

or replicating garbage collection techniques, local allocation heaps and so on), but the characteristic advantages of the thread-local heaps cannot be obtained for the memory blocks allocated in the shared heap. Furthermore, performing a garbage collection of the shared heap implies determining if there exists any path from local roots to the shared heap, therefore additional synchronization with the local heaps is necessary. In other terms, a garbage collection of the shared heap can be as expensive as it would be collecting a single, common heap.

That leads to two efficiency considerations: first of all, the more blocks are allocated in the local heaps, the better the performance (the more frequent the local garbage collections will be). Additionally, certain blocks, which are part of shared data structures, will become at some point during their life part of the shared heap. Some applications might rely on large structures which are continually accessed by several threads, and in that case a thread-local heap system might be of little help, but other applications might only occasionally need inter-thread communication. The performance gain, in this case, can be quite noticeable, especially if the cost of thread synchronization, for the chosen GC algorithm, is relevant.

11.1.3 Shareability by reachability

The definition of "shared", in practice, is open to interpretation. The simplest definition of shareability defines a heap block as shared if it is reachable from data structures that are common to more than one thread as, for instance, Java static variables and global roots. According to this definition, as soon as a pointer is created to a local block from a shared structure or a block already shared, the local block becomes shared, as do all the blocks transitively reachable from it. The actual implementation could materially move the blocks in a separate memory area, or just flag the block as "shared" without moving them. Since it is a pointer write that can cause blocks to change heap, the possible implementations will typically involve a write barrier of some sort.

11.1.4 Static analysis

An alternative definition of "shared," for memory blocks, can be obtained if a static analysis of the program code is performed. As shown by Ruf [Ruf00] and Steensgaard [Ste00], a static escape analysis can be performed to discover those blocks that, despite being possibly reachable from multiple threads, are in reality only ever used by a single thread, and can be consequently safely allocated in the local heap. The blocks that appear, with the static analysis, to be potentially used by more threads are instead all preallocated in the shared heap, and no block is dynamically moved from the local to the shared heap.

While the technique looks appealing, no dynamic profiling is used and a purely static analysis is used while deciding that a certain block should be preallocated in the shared heap. That could cause more blocks than necessary to be classified as shared rather than local, reducing the potential gain obtainable from a thread-local heaps system. On the other hand, a dynamic mechanism as the one previously described, that relies only on the notion of reachability, might find that certain memory blocks are reachable by shared structures even though the escape analysis could determine that they are actually only used by a single thread. In such cases, a block allocated in a

local heap would be moved into the shared heap, according to reachability, unnecessarily. Interestingly, the two approaches are not mutually incompatible, and could be combined, in principle, as suggested later in Appendix B.11. In the remained of this discussion, memory blocks will be considered to be shared if they are reachable by multiple threads.

11.2 Implementation alternatives

As previously mentioned, implementing shareability by reachability implies the use of some technique (usually a write barrier) in order to detect when a heap block becomes reachable from multiple threads, so that it can be properly flagged, or moved. The invariant that must be maintained is that no pointer is ever present pointing from the shared heap into a local heap. Since a reference to a local block can only become known to other threads if it "leaks" into the shared heap or a global root, it is sufficient for each thread to keep under control the pointers it creates from the shared heap, or a global root, into its own local heap. If the creation of such a pointer is attempted, the invariant would be invalidated, and a corrective action must be taken. In order to reestablish the invariant, the block must be moved into the shared heap, together with its transitive closure.

The use of write barriers is not the only possible option to detect the creation of new references from the shared heap into a local heap, but it appears to be the simplest and probably most efficient solution. Unless specialized hardware is used, the only alternative appears to be the use of the MMU to intercept all writes to the shared heap. In that case, however, all writes to the shared heap would cause a trap, which is likely to involve a very large overhead.

11.2.1 Copying vs. flagging

From the point of view of the implementation, the logical subdivision in multiple local heaps and a shared heap can be implemented in different ways. The allocated blocks can be, for instance, flagged as being logically in one heap or another, even if they physically reside in the same address range. Alternatively, the various heaps could be kept in different memory areas, and the blocks copied from a local heap to the shared heap when they become reachable by more than one thread.

Moving heap blocks involves finding all the pointers that point to the block that is being moved, and patching them with the new address. That is a rather expensive operation, and simply flagging the blocks, therefore, would appear at first a more convenient solution. However, there are a couple of important drawbacks. First of all, it is no longer possible to freely allocate independently blocks in local heaps without synchronization (unless local allocation heaps, as previously described in Section 11.1.1, are used to avoid most of the synchronisation). Additionally, it is not possible to perform a moving garbage collection of the memory used exclusively by a single thread, without synchronisation with the remaining threads, since all blocks physically reside in the same space. Finally, and more importantly, the cost of the write barrier can become *very* high, since it involves finding the flag local/shared of each of the two blocks and comparing them each time a pointer is written. In the paper by Domani et al. [DKL+02], their conclusion is that "the overall garbage collection time was cut on average by about 50%" using thread-local heaps.

However, the cost of their barrier is so high that “the improvement in collection was offset by the cost of the write barrier.”

11.2.2 Segregated heaps

The idea of moving the blocks to a different memory space, copying their content, is also summarily dismissed, in the same report, arguing that moving blocks out of the local space when they become shared (via the write barrier) “[...] requires knowledge of all the local references to the moved objects [...] Finding these references would be an unacceptable cost.” While it is true that the mentioned operation is expensive, there is no analysis in the paper of the real frequency with which such an operation would be necessary. In particular, if it is possible to predict correctly, using a profiling technique, the correct heap in which to preallocate most blocks, the number of times in which some blocks would need to be moved could be rather low.

If the local and shared heaps are kept segregated in different address ranges, an implementation of the write barrier can obtained with little more than the comparison of two pointers, and reduced to just one or two additional assembler instructions per pointer write in the heap. If, on average, the write barrier does not trigger further operations, and the number of block copies that would be required is low, having a shorter write barrier while sporadically moving physically heap blocks from the local to the shared heap could end up being a very convenient solution after all.

A study, conducted by the author on the applicability of this technique, seems to indicate that, being able to preallocate blocks in the right heap, the number of blocks that need to be moved can be quite modest. A summary of the study conducted, and the results obtained, is available in Appendix B. Implementing segregated heaps as described involves however some difficulties, which will be described in the following section. Pointer discovery will be the most straightforward solutions to those problems.

11.3 Pointer tracking: a practical solution

Implementing thread-local heaps using segregated heaps, as said, involves the use of write barriers (either using explicit checks or using the MMU to trap the accesses) that can intercept the creation of new references from the shared heap into a local heap. As a consequence of an attempt to write such a reference, the referenced block, and an arbitrarily large number of further blocks, might need to be moved from the local heap into the shared heap. The write barriers can be considerably simpler using segregated heaps than they would be using flags for each heap block. However, moving memory blocks on-the-fly (during a pointer write) involves some additional issues.

Since it is not possible to determine in advance which blocks will need to be moved, every time a pointer is written the content of the heaps might need adjustments. In other terms, all the pointers to heap blocks in the system could need to be adjusted, be they in the heap, in the stack or the registers, every time one of the threads performs a write operation involving a pointer. If the threads are scheduled preemptively, or if there are multiple microprocessors working simultaneously, all the remaining threads could be at arbitrary points in the code.

The set of techniques discussed in this Ph.D. research, however, are designed exactly to obtain the complete set of pointers active at any point in the code. Using the techniques described, it becomes therefore possible to move memory at every moment without imposing particular restrictions on the code. The general pointer discovery technology, consequently, is not only suitable to implementing known program services, but it is also a precious instrument that can enable the development of new technologies.

What a wonderfully witty quote!
*— **Antonio Cunei**, Ph.D. candidate*

Chapter 12

Evaluation and Conclusions

This thesis has explored the software infrastructure required to support program services like garbage collection, persistence and migration when using fully optimised native code, so that the various services can operate preemptively, at every machine instruction. An extensive discussion about the requisites, the complexities and the advantages of such support was followed by a description of the techniques that can be used to implement the desired support. Aspects already present in literature were analysed in much greater depth, and new techniques introduced.

Among the many technical issues discussed, this work has analysed:

- the common requirements for the preemptive execution of the mentioned program services
- the use of "PC maps" to store the necessary information for every machine instruction
- methods that can be used to determine where pointers are located in registers, for every machine instruction, including a particular form of liveness analysis
- issues and techniques involved in determining where pointers are in the stack and the heap
- the mechanisms used at runtime to obtain the data necessary to the service routine
- an experimental prototype used to expose some of the less obvious technical details, including the interaction of preemptive services with legacy environments, the fine details involved in register windows and delay slots, and the interface with an existing compiler system
- the issues involved in supporting fully optimised code, in particular techniques that can be used to deal with derived pointers
- the use of PC maps to support memory-management techniques, in particular segregated thread-local heaps

This research has originated some algorithms and techniques believed to be new, as will now be explained.

Contributions

The main contributions of this research are the following:

- A form of liveness algorithm has been developed that can be used to determine the state, out of a fixed set of possible states, of a register or variable at every instruction. Such information is calculated using local def/use information for each instruction, information that however is not trusted to be consistent. A series of consistency checks are shown to be possible using simple algorithms that have modest computational complexity. All of the algorithms are formally explained and justified in Chapter 5. A non-trivial extension to the case of Delayed Control Transfer (delay slots) is also investigated, and the consistency checks suitably adapted.
- The experimental prototype described in Chapter 9 is the first implementation described in literature able to generate PC maps from multiple source languages without requiring modifications of pre-existing front ends. This result shows that it is possible, at least to a very large extent, to decouple the language front ends from the actual generation of PC maps.
- A comprehensive and analytical compendium of concepts and techniques related to the implementation of PC maps has been developed in detail. The experience gained while analysing the problems and implementing a prototype has been documented in this thesis, and it will hopefully be of use to other researchers.
- An original approach to the handling of derived pointers, reducing them to internal pointers, is also believed to be described here for the first time. While the technique entails some additional work in the heap manager, arising from the need to support internal pointers, the idea appears to be worthy of further investigation. The use of derivation tables, in the style used by Diwan at al., remains a valid alternative.
- A further new element contained in this work is the applicability of PC maps defined on every instruction as a possible solution to the implementation of certain memory management techniques, in particular segregated thread-local heaps, as discussed in Section 11.2.2. More details on this aspect are available in Appendix B.

Evaluation

The use of PC maps can enable a service routine, like garbage collection or checkpointing, to intervene with a finer granularity than possible using additional instructions inserted at selected points in the code. Potentially, the approach can be used to inspect and alter the state of the memory at every machine instruction. In that respect, PC maps have the potential to reduce latency when a service routine is called. When preemptive thread scheduling is used, avoiding the need to roll forward all the threads to safe points can lead to a noticeable reduction in overhead when invoking service routines.

Certain aspects of the technique might however benefit from further investigation. In particular:

- The handling of derived pointers currently involves some trade-offs between the level of code optimisation and possible runtime overheads. Finding an implementation of the heap manager that can minimise such overheads would allow for a better support of derived pointers.
- In order to fully realise the potential of the technique, and to be able to interrupt all the threads at arbitrary moments, the use of PC maps should be extended to libraries and system components, which is difficult to do in a legacy host system. Improving the cooperation between user code and existing legacy code would be important to improve efficiency in a system, hosted by a conventional programming environment, that relies on PC maps to handle service routines.

On the other hand, definite positive aspects also emerged from the analysis and the test implementation. For instance:

- Implementing PC maps, at every machine instruction, was shown not to be a problem even in the presence of unusual microprocessor features like register windows and delay slots. It has been shown that multiple programming languages can be supported using PC maps within the same compiler system.
- The same infrastructure, offering detailed information about pointers, can also be used to support memory management techniques that require memory to be moved on-the-fly.
- The size of the additional tables and data structures needed to support preemptive service routines is generally quite modest.
- While supporting certain features of programming languages can be more or less challenging, none among the most fundamental structural features of programming languages was found to be an inherent obstacle to the use of PC maps, and consequently to preemptive program services.

In conclusion the technique has good potential, but certain aspects could be improved with further study, in particular offering an efficient support to derived pointers while using fully optimised code. Synchronisation with a legacy host system can also be challenging, and more work could be devoted to that aspect. The use of PC maps with fully optimised code, especially if derived pointers are required, might be somewhat premature for implementation in production systems at this stage.

Nonetheless, the technique is indeed promising and, as extensively discussed, there are no major technical obstacles to its implementation. Furthermore, the information supplied by PC maps can be used to implement forms of preemptive memory manipulation that would otherwise be unavailable. As to the advantages that can be obtained with respect to existing systems based on safe points, a definite answer can only be obtained by implementing a more complete prototype, able to support most of the features required to run real-life programs. The real measure of the benefits that can be achieved also depends, to some extent, on the ability to address efficiently the aspects previously described. While some ideas have been suggested in this thesis as to possible paths to follow, there is plenty of scope for more research, and for further implementation ideas to be developed.

Future developments

There are several immediate developments that can follow this work. Some of them are listed below.

- Implementing preemptive support for program services, using optimised code and also supporting derived pointers, is possible using the techniques described in Chapter 10. At the moment, however, the systems described in the literature are only able to support either derived pointers but not preemption [DMH92], or preemption but not derived pointers [SLC99]. An extension of the prototype described in this thesis in order to support derived pointers preemptively would be the first system to support all the named features while using optimised code. In particular the techniques described by Diwan were implemented on the same platform as the prototype described in this work (the GCC suite), which should simplify a porting effort.
- One of the relevant aspects of this work is the separation between the various service routines and the rest of the system, enabling the use of different services with the same code. That should enable, for instance, preemptive checkpointing in a persistent system. So far, there does not seem to be any report of systems implementing any kind of service preemptively except garbage collection. A straightforward development of the current work could be writing a simple service routine implementing checkpointing, for instance, and verifying that the operation can indeed be executed preemptively as expected.
- While this work has focused primarily on garbage collection, preemption and migration, other program services can also benefit from the general technique. A possible development would be to explore the requirements, and the possibly more complex infrastructure, necessary to support further program services. Exception handling, for instance, is not described in this work, but its integration into the system described might be useful. A paper by Chase describes some of the problems related to the preemptive treatment of exceptions [Cha94]. Debugging can also be considered, to a certain extent, as a program service, and a discussion on the subject can be found in the documentation related to the `C--` system [JRR99].
- In the longer term, the prototype could be made more robust and ported onto different microprocessor architectures. Implementing more features useful for a production system, as discussed in Section 9.9, would enable the system to become a more complete programming toolkit, able to support transparently preemptive program services with a more extensive range of real-life programs and programming languages. An appealing feature of such a toolkit would be the ability to easily add and replace service routines without altering the compiled user code, thanks to the standard interface between the runtime core and the various services.

Conclusions

The use of PC maps can effectively support preemptive program services, and the same technique can be of help while developing memory management techniques. While certain issues remain open (an improved support for derived pointers, for instance), there is plenty of promise in the overall technique. As a consequence of the experience gained in this work, it is also possible to strongly advocate the inclusion of a clear separation among multiple primitive data types, down to the level of the assembly code, in the core structure of compilers. The feature can be used to obtain more detailed information about the data manipulated by the final compiled code, simplifying the implementation of services like garbage collection and persistence.

It is the hope of the author that this work can be a useful reference to those who might want to explore the subject further, and possibly devise additional techniques. The time spent working on it was, for me, a continuous discovery of technical, human, and scientific aspects. It was also a humbling and enlightening lesson about the meaning of research, and it certainly taught me a great deal about how to develop a research project. It is this last aspect, that I personally find the most important result obtained.

Writing code is an act of creativity.
It isn't science and it isn't engineering.
*— **Bryan Dollery**, DevX.com*

Appendix A

JBE and ExactVM

The JBE Java compiler was developed by Sun Microsystems as the optimising compiler of ExactVM. During my internship in Sun Microsystems Laboratories in Mountain View, California, I discovered, thanks to Mario Wolczko, that the internal structure of JBE made use of techniques that are closely related to the ones described in the present work. Although there is no publicly available literature describing the architecture of JBE, Ross Knippel, David Cox and Chuck Rasbold kindly explained to me in detail some of the internals of their compiler. What follows is a short report based on an extremely interesting meeting I had with them at the Sun campus in Santa Clara, CA.

The origin of the project lies in a previous compiler named UBE (Universal Back End), which had multiple front-ends for FORTRAN, C and C++. The focus of the project later shifted on the Java language; the previous front-ends were replaced by a Java front-end and support for exact garbage collection was added. The compiler was renamed JBE, and was integrated into Sun's ExactVM (also known as ResearchVM), together with a basic Java interpreter and a simpler JIT compiler.

The way in which JBE achieved exact garbage collection was through the creation of tables, in the form of bitmasks, that allowed the system to identify, for specific locations in the compiled code, which registers and stack locations contained references. Although the tables were generated only for specific points in the code, according to their description of the system, the tables could have been generated, potentially, for every machine instruction. JBE contained therefore, at least to some extent, the compiler infrastructure that would have allowed garbage collection and other memory operations to be performed preemptively.

The information contained in the tables was used to quickly obtain the mode (reference or non-reference) of each register and each stack slot. Since those tables were accessed very often, no attempt was made to compress them. A static liveness analysis was performed to locate the pointers in registers and stack slots. The ambiguities in the use of stack slots were resolved by

modifying, with a separate pre-pass, the stack offsets so that no stack slot could be used as both a scalar and a reference within the same routine.

Offering support exclusively to Java, however, allowed the developers to avoid some of the traps described elsewhere in this work. For instance, only pointers to the initial locations of objects were required. Arrays can never be contained in the stack; therefore, the mode of each stack slot can be determined by a simple liveness analysis (see section 3.8 for further information). Because of the structure of the compiler, it was never necessary to deal with split pointers.

There cannot be a crisis next week. My schedule is already full.
*— **Henry Kissinger**, 1923 -*

Appendix B

Preallocation in Segregated Thread-local Heaps

As discussed in Section 11.2.2, the use of distinct memory address ranges to store the shared and the local heaps in a thread-local heap system allows the implementation of the write barrier to be reduced to very few machine instructions. However, copying objects from the local heap to the shared heap is a potentially very expensive operation. In order to reduce the need for object copying, it is desirable to preallocate the objects, if possible, directly in the most appropriate heap (shared or local) so that no further object relocations are required.

However, it is not easy to decide when to preallocate an object in the shared heap. We have previously mentioned that a static analysis can be used to decide in advance when certain objects are certainly local. However, the use of some dynamic information, obtained from an actual program run, could offer a better indication of the heap in which to preallocate objects. In this appendix, some original research will be presented about the use of dynamic profiling in the context of a thread-local heap system, implemented using distinct address spaces for the various heaps.

B.1 The call chain as an indicator

A simple, but very effective, method to determine whether an object is likely to become shared or not, is to verify what happens to all the objects previously allocated at a certain allocation site, possibly using, as a context, the last part of the dynamic call chain that led to that point in the program. The overall idea is that some parts of the program are enclosed in loops that perform similar operations repeatedly. If an object is allocated in a point of the program that is reached through a certain call chain, and that object becomes shared, it might happen that other objects, allocated through the same call chain, follow a similar history.

The first step in the analysis can be to verify whether this hypothesis holds, so that we can use the call chain (or part of it) as an indication of the likelihood of an object to become shared. Similar studies have been conducted, with interesting results, in the context of generational garbage collection, trying to preallocate objects likely to have a long lifetime directly in the old generation. In a paper by Cheng et al. [CHL98], the authors identified allocation sites likely to allocate long-lived objects by performing a preliminary run of the program, from which some profiling information is obtained. While more detailed information is available after the completion of a test run, the need to proceed with a preliminary, separate data-gathering run may be inconvenient. Domani et al. [DKL+02] applied a similar technique, opportunely adapted, to thread-local heaps. In their work, a statistical analysis is performed at runtime in order to discover which allocations sites are most frequently used to allocate blocks that eventually become shared. Those sites are then modified in order to preallocate blocks directly in the shared heap. Their choice was to gather the necessary data using a preliminary test run, and proceed with the analysis only afterwards.

B.2 Dynamic profiling

An alternative way to obtain data concerning the allocation sites could be the use of dynamic profiling techniques. The idea is to use information gathered dynamically on the past program behaviour to make predictions about its future. Such predictions are then used to adjust some execution parameters on-the-fly, hopefully improving performance. Harris [Har01] explored the feasibility of dynamic profiling applied to pre-tenuring, using the last part of the call chain to make decisions about which objects to preallocate in the old generation space without the need for a separate data-gathering run.

The missing step is trying to explore the efficacy of dynamic profiling used to create predictors in the context of thread local heap. While it is true that the final condition of a block (local or shared) cannot be known for certain until the block eventually becomes shared or dies, it may be interesting to explore whether the transient behaviour of past blocks allocated at a given allocation site can be used as a reliable indicator of what will happen to other blocks. If that is the case, a statistical analysis could be used to decide dynamically whether to preallocate further blocks, allocated at the same site, directly in the shared heap. The use of such a dynamic profiling, applied to thread-local heaps, was seemingly never applied in literature. It will be shown that, for some programs, even a very simple profiling technique (that involves very little overhead on the normal program execution) is sufficient to make surprisingly accurate predictions.

B.3 Correlating allocation sites and shareability

As previously mentioned, a possible way of predicting whether a heap block that is being allocated will become shared or not is to verify whether there is a correlation between that condition and the allocation site. We will use the term “allocation site” in a general sense to refer to the combination of the actual position in the code where the allocation is requested, in conjunction with the most recent portion of the dynamic call chain. If we can show that there is a reasonable correlation

between shareability and allocation sites, then it is fair to assume that the allocation site can be used as a sufficiently reliable indication of whether it would be convenient to preallocate certain objects in the shared heap or not.

The use of allocation sites to preallocate objects has been described extensively in the context of generational garbage collectors. For example, the already mentioned Cheng et al. [CHL98] gather information about allocation sites likely to allocate tenured objects from preliminary test runs and Harris [Har01] gathers similar information dynamically. Using of allocation sites as predictors to preallocate objects in a thread-local heap system has been described by Domani [DKL+02]. The criterion they used was to consider likely to allocate shared memory blocks those allocation sites in which, in the test run, at least 99% of the allocated blocks eventually became shared. Their good results show empirically that a correlation between allocation sites and shareability exists. However, it is not immediate to understand how strong that correlation really is, and it is actually not easy to represent that correlation in a clear numerical or visual form.

A useful way to visually correlate allocation sites and objects having a certain property is used by Harris, who uses a particular type of diagram to illustrate how allocation sites are correlated to objects tenured or non-tenured, in the context of a generational garbage collector. The same kind of graph can easily be adapted to the context of thread-local heaps. Therefore, before proceeding with the details of the prediction techniques, it can be useful to show concretely the kind of correlation that exists, so that the use of allocation sites to predict shareability is actually justified.

B.4 Percentage of objects vs. categories

The way in which the graph is to be interpreted is the following. Let us assume we have a number n of items, each of which has a certain numeric property, represented as a number between 0 and 1. If we desire to have a visual indication of the distribution of that property among the items, we can imagine dividing the segment [0,1] on the x axis in n equal parts, one per object. For each item, we plot the value of the property as a number in [0,1] along the y-axes, while the x value is the value along the x-axis assigned to that item. The resulting graph, in general, will alternate points with high and low y values in a non very intelligible way. However, if we pre-sort the items according to the value of the property before plotting the graph, the result will be a visually clear representation of what proportion of items has what value for the property in question.

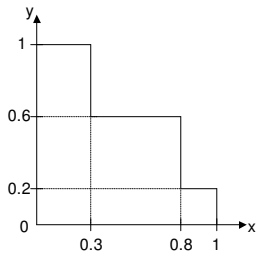


Figure B.4.1: Distribution graph

For instance, in Figure B.4.1, 30% of the items have the property set to 1, 50% to 0.6, and 20% to 0.2. We can easily use the same graph to depict what percentage of blocks become shared for a certain allocation site. We will use as “items” the allocations sites and as “property” the percentage of blocks, allocated in that allocation site, that become shared. In the ideal case, we

would like to see sites with a percentage of either 100% or 0% of the blocks shared, so that it is possible to separate completely allocation sites that allocate local blocks from allocation sites that allocate shared objects. Comparing that ideal case to the reality of some test benchmarks will give us an idea of how good that correlation is.

B.5 Correlation graphs

In order to gather the necessary information, some traces were obtained from an adapted version of the Tracing Java Virtual Machine [Wol99], an instrumented Java VM capable of tracking the complete history of all the allocations and modifications applied to objects. The traces recorded contained the value of the program counter and the last elements of the dynamic call chain. Analyzing the trace files, the shared/non-shared condition of each object was reconstructed, and the following graphs were generated, in which the correlation between the shared/local condition and the allocation site is shown.

The obtained correlation graphs, related to different test programs, are shown in Figures B.12.1 and following, on pages 195–199. For additional convenience, a scaled version of Figure B.12.3(a) is shown here. The benchmarks used are explained later, in Section B.9. The different lines shown in each diagram represent the correlation that exists between the allocation sites (on the y axis) and the percentage of objects, allocated in that site, that eventually become shared (on the x axis). Each line refers to a different portion of the dynamic call chain used when determining the allocation sites. The graphs show that the greater the portion of the dynamic call chain is considered, the stronger the correlation becomes.

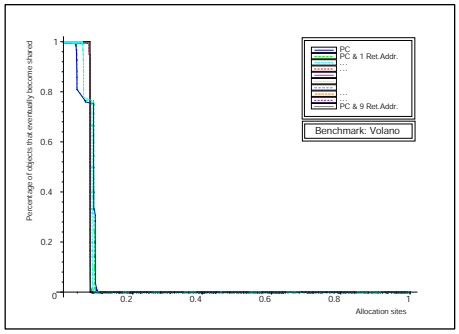


Figure B.5.1: Correlation graph: Volano

It is visually apparent that the allocation sites do have a strong correlation with the shared/local condition of objects. In certain cases, some allocation sites allocate both shared and local objects, presumably because of different lives of objects allocated there, during different stages of the program execution. Nonetheless, the use of allocation sites as indicators of shareability appears to be overall reasonable. Before concluding that the allocation site of an object can be reasonably used as a predictor, however, there is another detail that should be checked, as explained in the next section.

B.6 Delay graphs

The correlation graphs, discussed in the previous section, represent the ideal case in which the final condition of all objects (shared or not) is known in advance. Such information, unfortunately, is not fully available during a dynamic profiling, which can only rely on the data collected up to a

certain point during execution. That would not be a particular problem for generational collectors, since all objects are promoted to the old generation when they reach a certain age, and it is not necessary to wait the end of the program to determine whether the object was promoted or not. Conversely, an object allocated in a thread-local heap can remain local for an arbitrarily long time before becoming shared. The partial information obtained during execution could therefore be misleading, identifying as local an object which will eventually become shared.

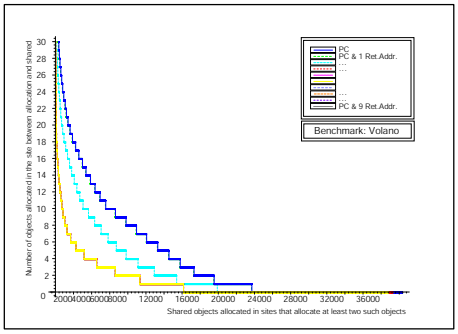


Figure B.6.1: Delay graph: Volano

In order to have an idea of the actual delay between the allocation of an object and its transition to the shared state, a different graph is constructed. In the delay graphs shown in Figures B.12.1 and following, all the objects that become shared, and which are allocated in an allocation sites that allocates at least two objects, are distributed on the x axis. The corresponding y value is the number of further objects allocated, in the same allocation site, between the allocation of that object and the moment in which it becomes shared. The objects are sorted, on the x axis, according to their value on the y axis, so that the resulting graph is monotonic. Figure B.12.3(b) is copied here, in a reduced size.

The graph shows how many other objects are allocated, in the same site, between the creation of a given object and the moment in which the same becomes shared. The higher the number, the greater the lack of precision in the prediction that relies on objects previously allocated in the same site, since some of those objects appear to be local, while in reality they will become eventually shared.

Although the results vary, depending on the specific benchmark, the tests suggest that the delay is usually rather low, especially if greater portions of the dynamic call chain are used to determine the allocation site. That means that most of the blocks destined to be shared do become shared after just a limited number of other blocks are allocated in the same site. As a consequence, after the initial stage we can assume that, on average, most objects destined to become shared will be recognisable as shared even using a dynamic profiling. Naturally, the greater the number of objects allocated in a certain allocation site, the higher the chance that older objects have become shared in the meantime. It can be expected, therefore, that the precision of the prediction will increase over time.

Summarising, the delay graphs give a general indication that dynamic profiling should be usable to predict shared objects with a reasonable degree of accuracy. A test of the usefulness of dynamic predictors, described later, seems to validate to a large extent the use of dynamic profiling for thread-local heaps.

B.7 Traps & Copies

As previously discussed, a system which uses segregated thread-local heaps would have the following characteristics:

- one local heap per thread
- one shared heap accessible from all threads
- each of the mentioned heaps uses a unique addressing range, separated from the others (segregated heaps)
- each local heap contains only blocks that are reachable from the corresponding thread. If a block becomes referenced from a shared structure, that block and all those reachable from it are moved in the shared heap
- the creation of new references is monitored using write barriers, so that new references from the shared heap into local heaps can be detected

In accord with this structure, the factors that will affect the efficiency of the system will be:

1. the ability to write a short write barrier
2. the number of occurrences in which a reference write from shared to local is detected, in proportion to all the reference writes
3. the proportion of blocks that need to be copied as a consequence
4. the proportion of blocks that are allocated in the local heap and are never moved from there

A write barrier will be executed each time a pointer is written in the heap. Keeping the execution time of the barrier as low as possible will be therefore very important. Also, if the reference writes from shared to local are only a small fraction of all the pointer writes, it is possible to use, in the write barrier design, the use of MMU traps. Even though serving a hardware trap is in general expensive, if we are able to design the write barrier so that it is on average shorter, and the trap is only rarely served, the overall gain can be considerable. In the remainder of this discussion, the term “trap” will be used to refer the detection of a pointer write from shared to local to indicate that, if the MMU is used to detect the pointer write, an exception is generated and a (relatively expensive) trap handler is called. The write barrier can be also implemented, however, using a simple sequence of test and branch.

The number of memory blocks that need to be copied from a local heap to the shared heap is also important for efficiency. As detailed in the introduction, moving a block is quite expensive, since all the references to the block which is being moved need to be found and updated. On the other hand, only the local heap that contains the block can contain references to it.

In addition to the parameters listed above, there are additionally considerations that should be made about the efficiency of the system. If everything was to be preallocated in the shared heaps,

for instance, the write barrier would never trigger a copy and there would never be any need to copy memory blocks. However, none of the blocks would ever be allocated in the thread-local heaps, and the advantage of performing separate garbage collections would be lost. Consequently, it is also important to try and maximize the number of blocks that can spend their lifetime entirely within a local heap.

As a final note, it is important to point out that deciding to preallocate a block in the shared space affects, in turn, the allocation of other blocks. All the blocks that are referred by the preallocated block will in turn become shared, even though, without preallocation, they might have lived initially, for some time, as local. Furthermore, the prediction might be wrong in certain cases, and some blocks that should have remained local might end up being shared. A certain level of error, since the predictor is based on incomplete data, is unavoidable. The results reported later, however, seem to indicate that those effects are, in general, fairly limited.

B.8 Gathering data

As mentioned, the traces obtained thanks to the Tracing JavaVM were post-processed to simulate the behaviour of a heap system structured around segregated thread-local heaps. Although that kind of simulation is not designed to give precise timing information as a complete implementation would do, an analysis of the traces can be used to establish some of the values described above. In particular, running the same trace file through different simulations can allow us to compare the efficiency of different prediction techniques.

B.8.1 Without prediction

A first test, useful for further comparisons, consists in the simulation of a thread-local heap system, for each of the trace files, without performing any preallocation. That gives an idea, at the end of the simulation, of exactly how many blocks, out of the total, remain reachable from a single thread. We will then compare that number with the number of blocks that remain local in the system when a predictor is used.

B.8.2 Allocation sites

The main set of predictors, used in the simulations, grouped the allocated blocks in classes according to their allocation sites. More precisely, two blocks are in the same class if the value of the program counter at the time of allocation is identical, together with the last part of the dynamic call chain. For each class, the number of blocks allocated in that allocation site and the percentage of blocks that have become shared so far are used to predict the likelihood that a new block, allocated in the same call site, will become shared. Some simple tests were performed, varying the depth of the portion of the call chain considered, the number of blocks that must be allocated before the predictor starts operating (in the initial phase there is no preallocation), and the percentage of shared blocks necessary for the predictor to decide that future blocks must be

preallocated in the shared heap. To give some examples, a predictor could start preallocating in the shared heap when, for certain allocation sites:

- using PC and 1 stack value, when, after 5 blocks allocated, 50% are shared
- using only the PC, when, after at least 15 blocks allocated, 95% are shared
- using PC and 2 stack values, when, after 45 blocks allocated, 90% are shared
- using PC and 6 stack values, when, after 5 blocks allocated, 60% are shared

While there are several combinations of parameters that can be used, and, as we will see in a moment, different way of structuring a predictor, performing simple tests like the above can give a quick idea of the kind of results that is possible to obtain.

B.8.3 Hashing predictor: simple but effective

One possible argument against the use of dynamic profiling is the overhead imposed by the execution of data-gathering activity while the program is running. Certainly, determining, as above, the current allocation site and comparing it against some sort of table or data structure, in which each site has distinct counters and statistics, is quite complex. For this reason, a much simplified predictor was used, trying to preserve some of the information obtainable from the allocation sites but at the same time trying to minimize the bookkeeping operations necessary to determine when to preallocate objects in the shared heap.

As an experiment, a hashing predictor was used, structured as follows: in a single table, an index is obtained using a simple hashing function that combines the PC and the last values of the call chain. For each position in the table, an integer counter is maintained, initialised to a predetermined negative value V. When a new block is allocated, its hash value (the value for its allocation site) is calculated and stored in an additional field of the block. Initially all blocks are preallocated in the local heap. For each block allocated as local, the matching counter is decremented by a constant C_L. If, as execution progresses, the object is copied from the local heap into the shared heap, the corresponding value in the table is incremented by the sum of the previous constant C_L plus another constant C_S, so that, as more objects become shared the counter gets closer to positive values, while for each object allocated as local the counter is progressively decremented. The predictor starts to preallocate objects as shared if the associated counter, for that allocation site, is greater than zero. The simple hashing function used consists in the lower bits (depending on the size of the table) of an exclusive “or” between the PC and the last return address. An exclusive “or” of PC and the last two return addresses has also been evaluated, as has been an exclusive “or” of the return address, shifted right, with the PC.

Such a predictor is rather crude, and in general it might happen that local and shared objects allocated in different allocation sites interfere with each other, causing erroneous predictions. The number of different allocation sites that is necessary to encode using the hash function can be greater or smaller, affecting the performance of this solution. However, in practical terms, the tests conducted showed a remarkable effectiveness of this simplified predictor, with overall results

not too distant from the predictor previously described, more complete but more complex to implement. Maintaining the simple mechanisms described for the hashing predictor is extremely easy, and can be implemented with really minimal overhead, making the idea of using dynamic profiling even more attractive.

B.8.4 Additional considerations

In the above predictors, the decision to preallocate in the shared heap, once taken, is never reversed. It might happen, however, that the prediction can be erroneous for many blocks if, in the program, to a first phase in which an allocation site generates shared blocks follows a second phase in which the same site allocates local objects. Those blocks would be needlessly preallocated as shared.

Reversing the decision, however, is rather problematic. Once the system begins to preallocate the blocks corresponding to a certain allocation site in the shared heap, the information originally used by the predictor becomes biased and no longer useful A possible way to discover whether an allocation site should return to local allocations could be to allocate occasionally, even if the predictor says otherwise, one of the blocks as local and check whether it becomes shared after a while. However, the interdependencies of the various blocks, and the fact that some blocks could have been needlessly allocated in the shared heap already, could cause the block to become shared because of the preallocation of previous blocks.

Another alternative could be to reset the predictor statistics, for each call site, after a certain number of blocks predicted to be shared, or even to reset the predictor for all the call sites simultaneously. Once again, the interdependencies among objects could sometimes cause the newly allocate blocks to become shared unnecessarily. Conversely, restarting the predictor would mean that some objects, which should be preallocated as shared, are instead allocated as local while enough new data for the predictors is gathered. No specific study was made in this analysis of predictors able to reverse their decision.

B.9 Some results

Some trace files, obtained from simple benchmarks, were processed, and the behaviour of various predictors tested. Because of the test environment used, it was somewhat difficult to obtain traces of heavily multithreaded functions. A test run for the Volano benchmark, which simulates a chat room server application, was nonetheless successfully analysed. During execution, the test run of the server portion of Volano created 412 threads, offering a good example of heavily multithreaded application to study.

Other benchmarks used were the 227 (mtrt) and 213 (javac) benchmarks from SpecJVM98, and other tests (“Pretzel” and “Paraffins” [PG02]) that create complex data structures. While these benchmarks are not inherently multithreaded, it is still possible to determine whether the objects are reachable from thread-specific structures or from shared ones (from global roots, for example). Although the results are not necessarily representative of what happens in massively multi-

threaded applications, studying those cases can give some indications of the program behaviours. Further simulations, in addition to those presented here, would probably be necessary in order to validate the efficacy of the studied predictors with other heavily multithreaded applications.

- Stats for: Objects are always allocated in the private space.
- At the end, there were 24,785 traps out of 34,335,990 pointer stores, that is 0%
- Excluding the pointer stores in the stack, which were 29,539,883, the traps were **24,785 out of 4,796,107**, 0%
- The traps so far caused **39,284 copies to the shared space out of 511,475** objects created so far, 7%
- Of all objects allocated, 511,475 were created in the private space and 0 in the shared space

Table B.1: Volano benchmarks, 412 threads. Objects always allocated in the private space

- **S**tats for: predict using PC and 1 stack level when, after at least 5 objs allocated for that site, 50% of them have become shareable.
- At the end, there were 6,654 traps out of 34,335,990 pointer stores, that is 0%
- Excluding the pointer stores in the stack, which were 29,539,883, the traps were **6,654 out of 4,796,107**, 0%
- The traps so far caused **7,179 copies to the shared space out of 511,475** objects created so far, 1%
- Of all objects allocated, 472,455 were created in the private space and 39,020 in the shared space

84% of objects in the shared space were directly preallocated
82% were correctly predicted and preallocated, out of all preallocated
81% were correctly predicted and preallocated, out of all that should have been shared
17% were erroneously preallocated, out of all preallocated
14% were mispredicted, or had to be copied, out of all those eventually in shared space
98% remained in private space, out of those that should have been
It performed 26% the traps and 18% the object copies of the all-private allocation

All in all, this predictor predicted correctly **98%** of all objects.

Table B.2: Volano benchmarks, 412 threads. Predictor based on allocation sites

Tables B.1 and B.2 show some statistics obtained from the Volano benchmark. The values shown in Table B.1 refer to a simulation in which no preallocation is performed. Table B.2 shows the results when a prediction is made using the allocation sites.

- Stats for: Hashing predictor, table large 32768, start:-10 - increments:2/-1
- At the end, there were 6,578 traps out of 34,335,990 pointer stores, that is 0%
- Excluding the pointer stores in the stack, which were 29,539,883, the traps were **6,578 out of 4,796,107**, 0%
- The traps so far caused **7,356 copies to the shareable space out of 511,475** objects created so far, 1%
- Of all objects allocated, 473,204 were created in the private space and 38,271 in the shared space

83% of objects in the shared space were directly preallocated
83% were correctly predicted and preallocated, out of all preallocated
81% were correctly predicted and preallocated, out of all that should have been shared
16% were erroneously preallocated, out of all preallocated
13% were mispredicted, or had to be copied, out of all those eventually in shared space
98% remained in private space, out of those that should have been
It performed 26% the traps and 18% the object copies of the all-private allocation

All in all, this predictor predicted correctly **98%** of all objects.

Table B.3: Volano benchmarks, 412 threads. Hashing predictor

It can be noted that, using a predictor to preallocate objects in the local heap, the number of object copies required drops dramatically, while the majority of objects can still be allocated in the local heap and handled with local garbage collection, reducing the need for synchronization. Table B.3 shows an example of a particularly good hashing predictor, as described in section B.8.3. In this particular case, the results are comparable to the much more computationally expensive

general predictor. While adjusting the various predictor parameters affects to some extent the results, all the predictor tried predicted correctly a very high percentage of objects, usually more than 95%.

B.10 Conclusions

The statistics shown in the tables seem to indicate that, using some simple prediction techniques, it is indeed possible to reduce the number of copy operations required in a segregated thread-local heap system to a very manageable level. The resulting lower overhead has the potential to make the technique viable for some applications, especially considering the potential advantages, namely the reduced number of synchronizations while performing garbage collections and allocations, and the ability to use compacting garbage collection algorithms, thanks to the address segregation.

In this research the Volano benchmark was the only massively multithreaded example used (142 threads in the trace file analysed). While the application is admittedly well suited to a thread-local heap system (the various threads have limited interactions), the application is nonetheless representative of a real class of multithreaded programs. Those programs could benefit substantially, according to this analysis, from the implementation of a segregated thread-local heap system. It would be however necessary to experiment with further multithreaded applications to verify the behaviour of such a heap system in general. The preliminary results obtained so far, nonetheless, appear to be rather encouraging.

It should also be noted that the write barrier, which should monitor all pointer writes in the heap, is extremely similar to the write barrier that is normally necessary in generational garbage collectors. In that case as well it is necessary to take action upon the creation of a new reference from one of the spaces (the old generation) into another space (the new generation). It is rather simple, therefore, to combine the two write barriers into a single one, so that only pointer writes in certain directions are intercepted. A particular technique that can be used for this purpose was devised by the author as a side product of this study, and it is currently being evaluated by Sun Microsystems for a possible U.S. Patent Office application.

B.11 Further work

An interesting possibility, apparently unexplored so far, consists in the integration of the static analysis with the notion of “shared if reachable by multiple threads.” That can be done by forcing certain objects, determined statically to be always used by a single thread, to reside in a local heap even though they appear to be shared, being reachable by other shared blocks. In order to prevent the write barrier from moving those blocks to the shared heap, it is sufficient to flag the blocks specially. When a pointer is created that originates from a shared block and points to such a special block, the flag on the target can be checked and, if set, the object is not moved to the shared space, since it is known from the static analysis that it will only be used, in reality, by a single tread. Transitively, all the blocks that are reachable only from local or specially marked

objects will only be used, again, by the local thread, and they do not need to be moved to the shared space. The blocks that are transitively reachable from marked blocks, however, will have to be marked in turn as special.

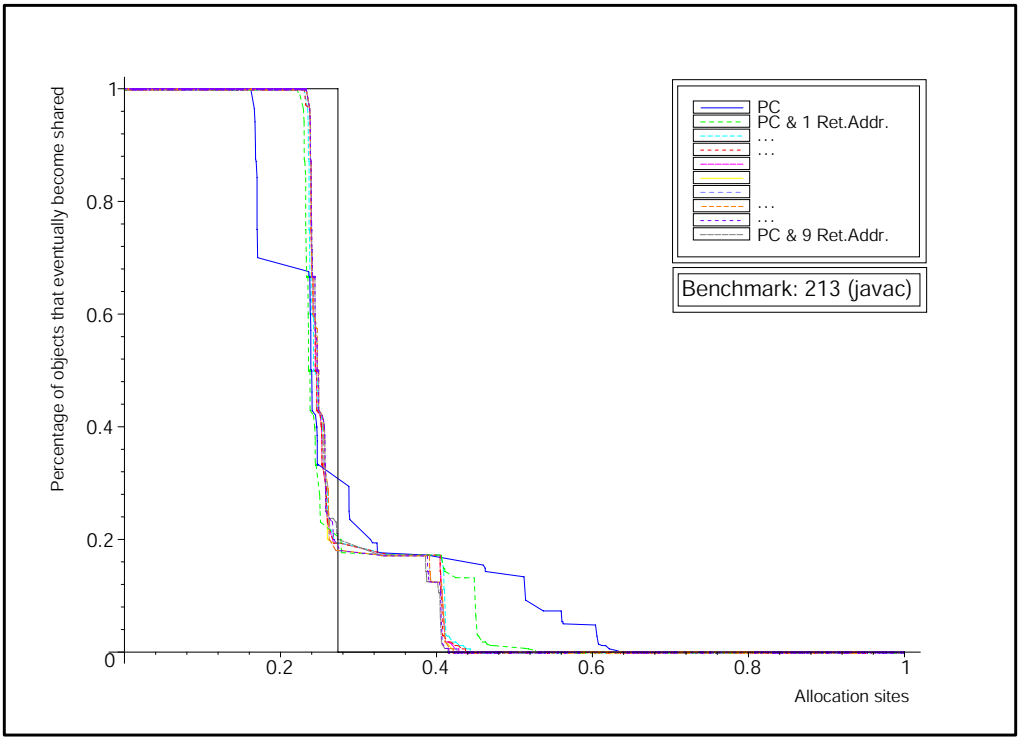
The use of such a hypothetical hybrid system could bring further improvements to a thread-local system that only uses reachability. No concrete implementations, or descriptions similar to the one here presented, is available in literature. A combined system of the kind described can certainly be implemented and analysed as a possible extension of the present work.

B.12 Graphs

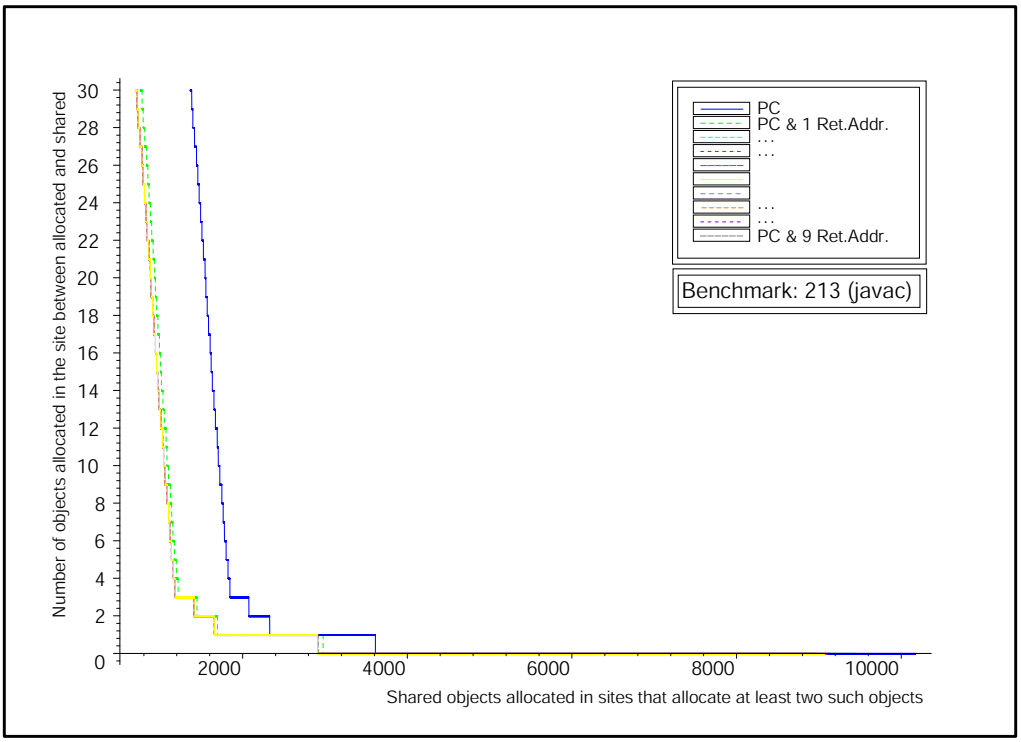
The following pages contain several graphs, one page for each benchmark. Each page contains two graphs. Each graph represents, using different line styles, the results obtained considering different portions of the dynamic call chain, in addition to the program counter. If *n* return addresses are considered, for instance, two allocation sites are considered distinct if at least one among the return address and the topmost *n* return addresses on the stack differ in the two cases.

On the top of each page, the correlation graphs, as explained in Section B.5, show the correlation between the percentage of objects that eventually become shared, on the y axis, and the allocation sites, distributed on the x axis and sorted according to the percentage of object that become shared at that site, so that the graph is monotonic (the number on the x axis represents the percentage of the total objects considered between that point and the origin).

On the bottom of each page, the delay graphs, as explained in Section B.6, show all the objects that become shared and that are allocated in an allocation sites that allocates at least two objects, distributed on the x axis (the number on the x axis shows the number of objects considered between that point and the origin). The corresponding y value is the number of further objects allocated, in the same allocation site, between the allocation of that object and the moment in which it becomes shared. The objects are sorted, on the x axis, according to their value on the y axis, so that the resulting graph is monotonic.

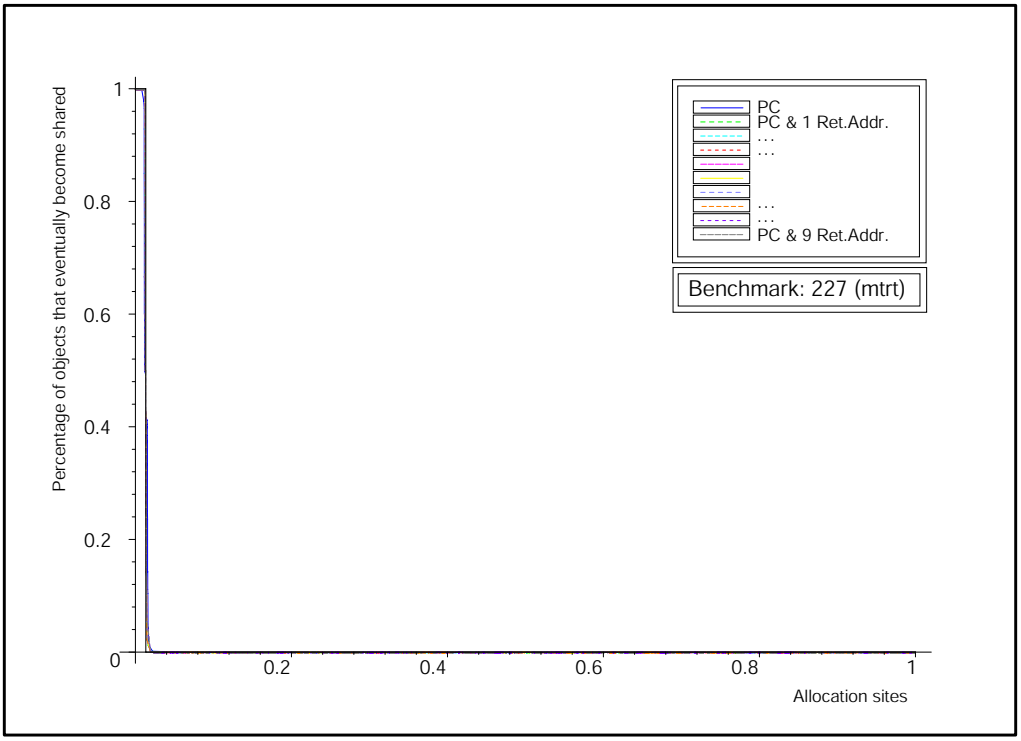


(a) Correlation graph

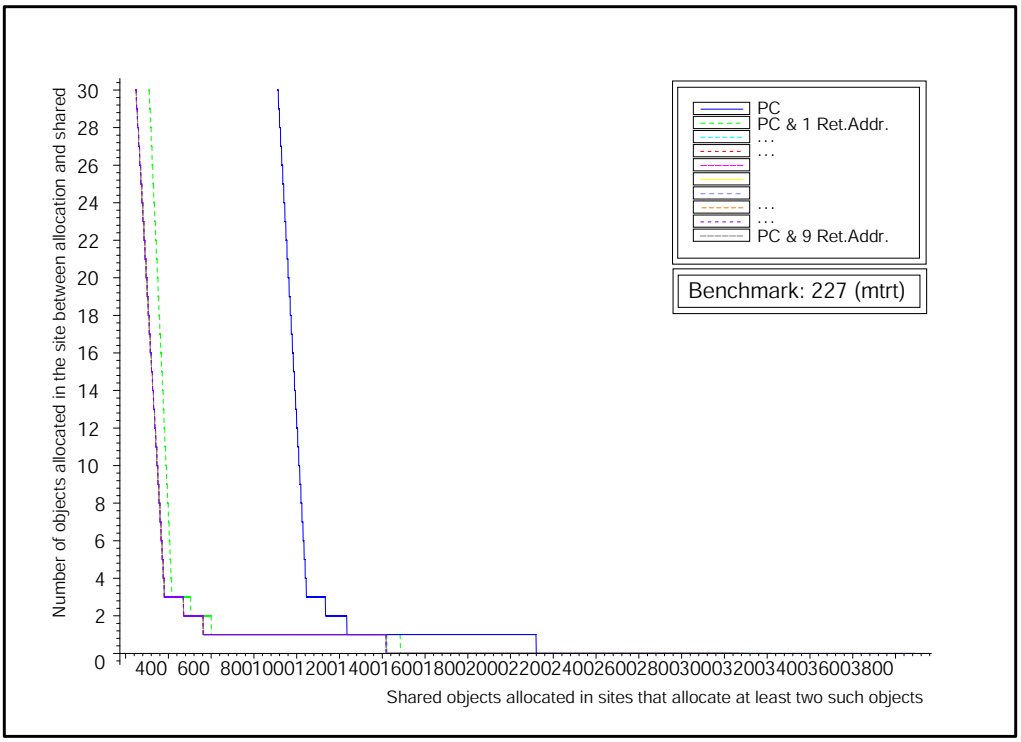


(b) Delay graph

Figure B.12.1: Benchmark: 213 (javac)

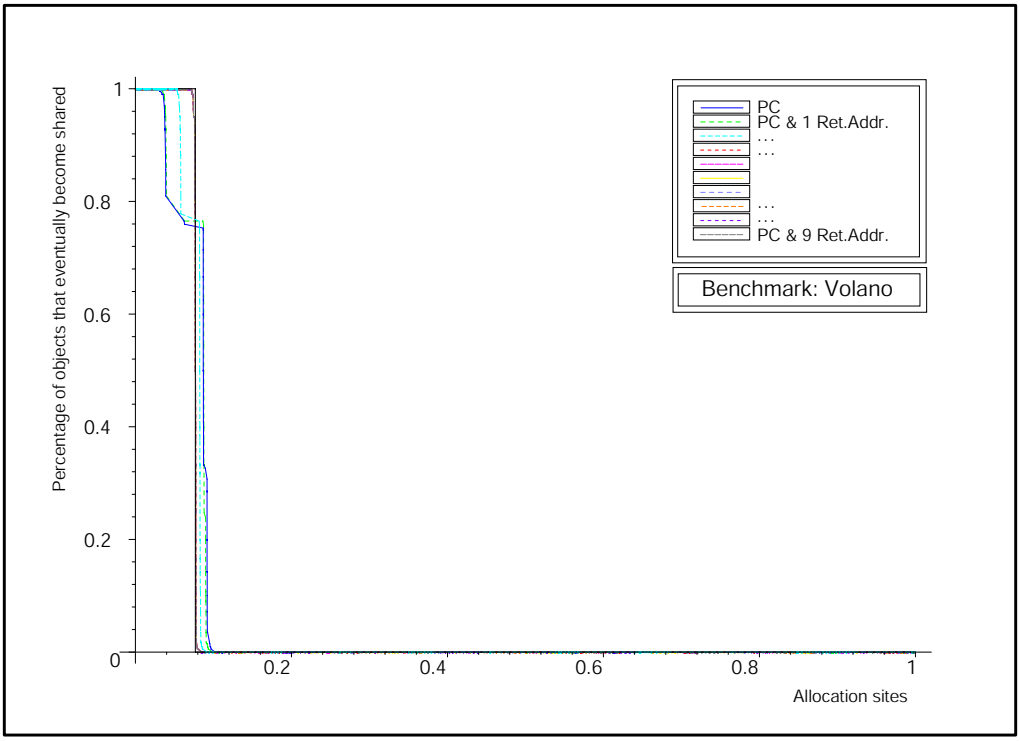


(a) Correlation graph

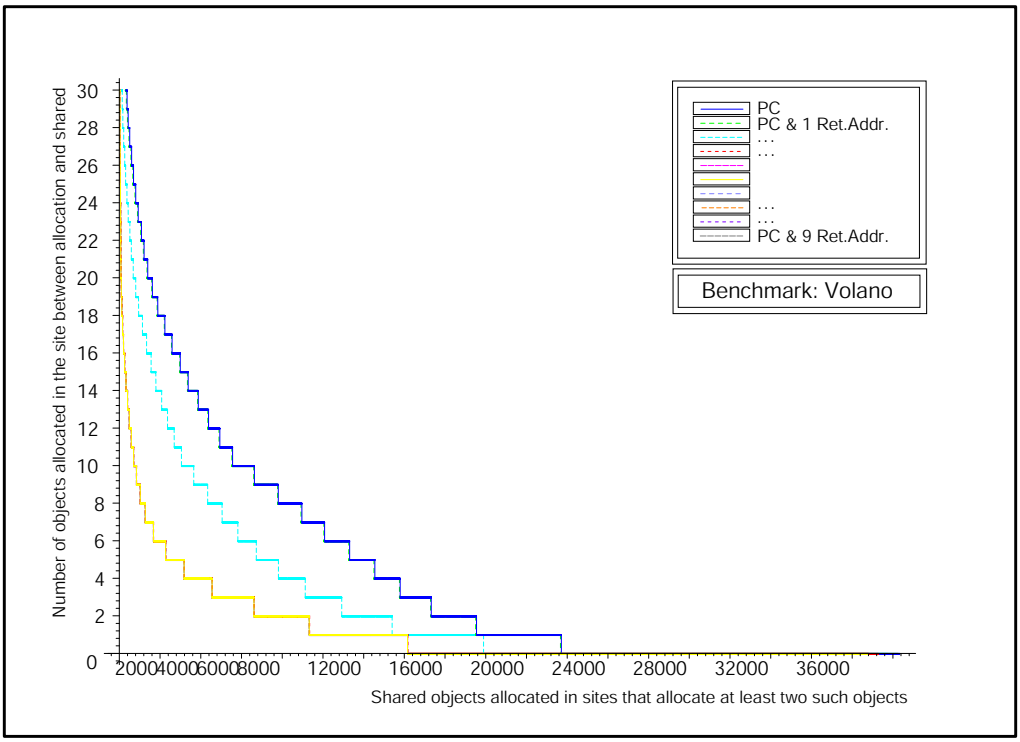


(b) Delay graph

Figure B.12.2: Benchmark: 227 (mtrt)

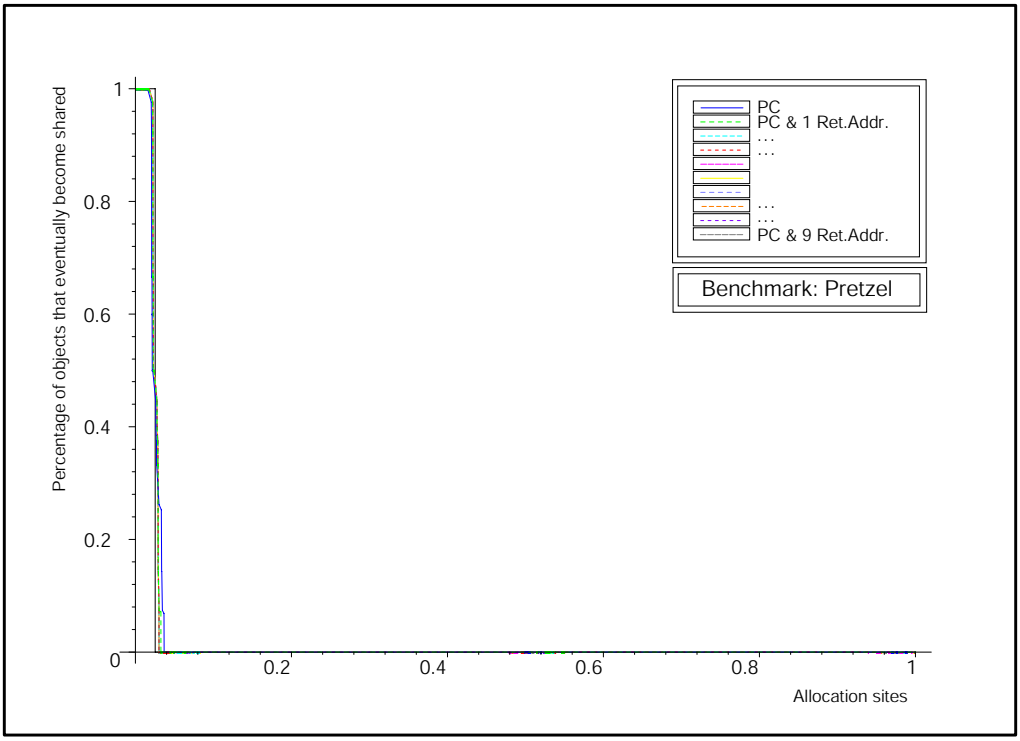


(a) Correlation graph

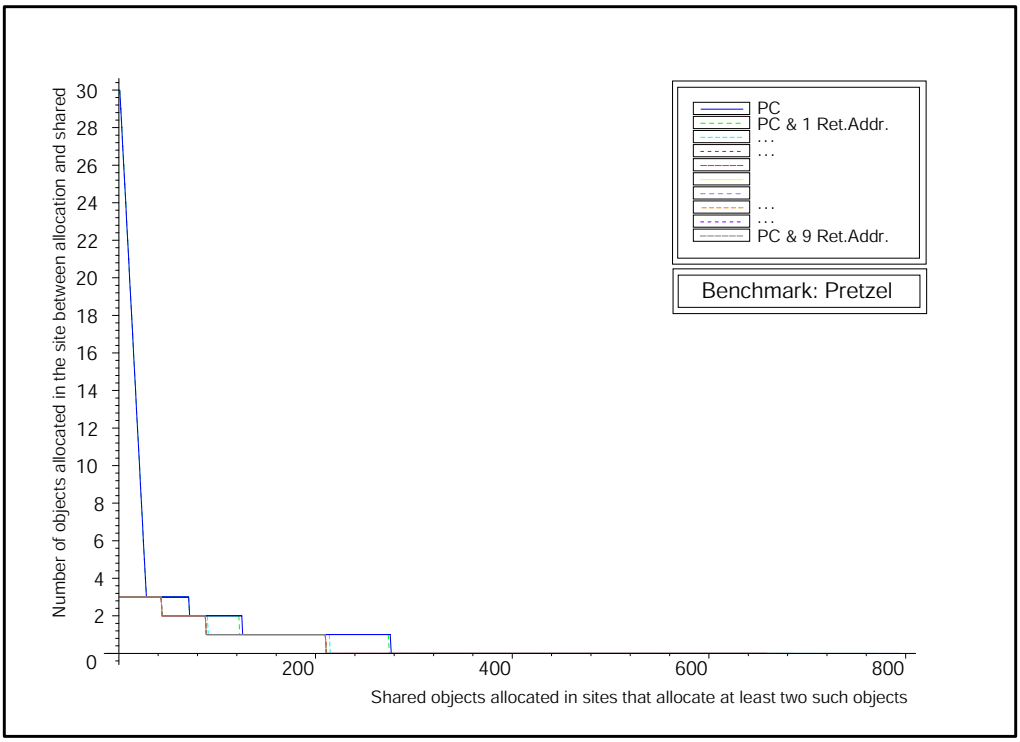


(b) Delay graph

Figure B.12.3: Benchmark: Volano server

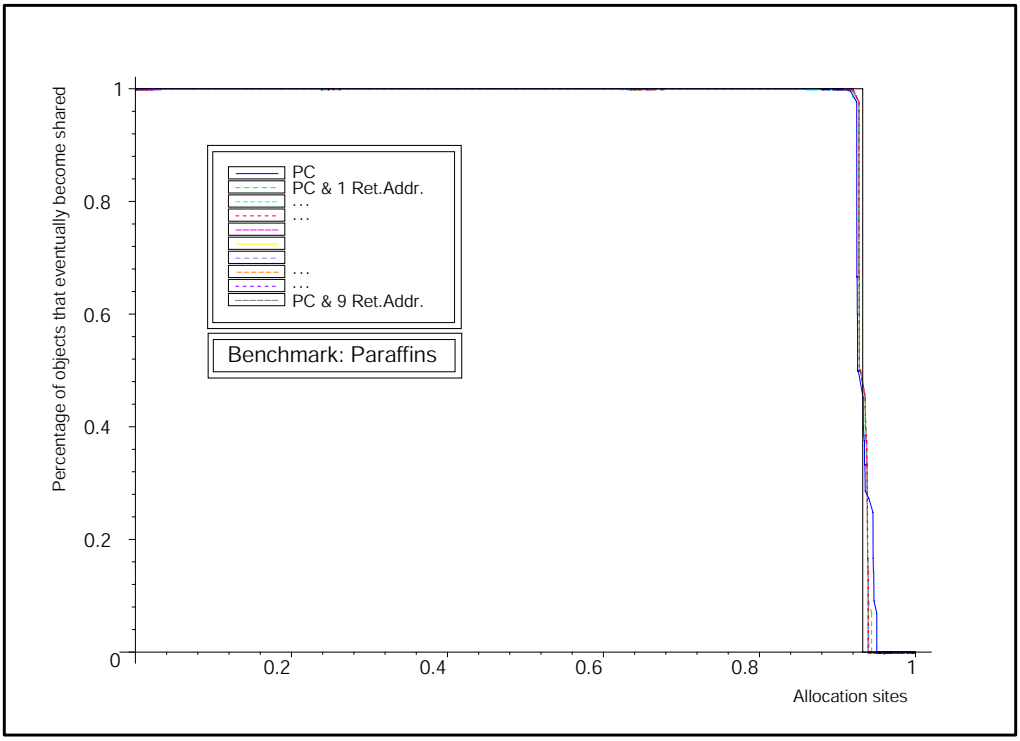


(a) Correlation graph

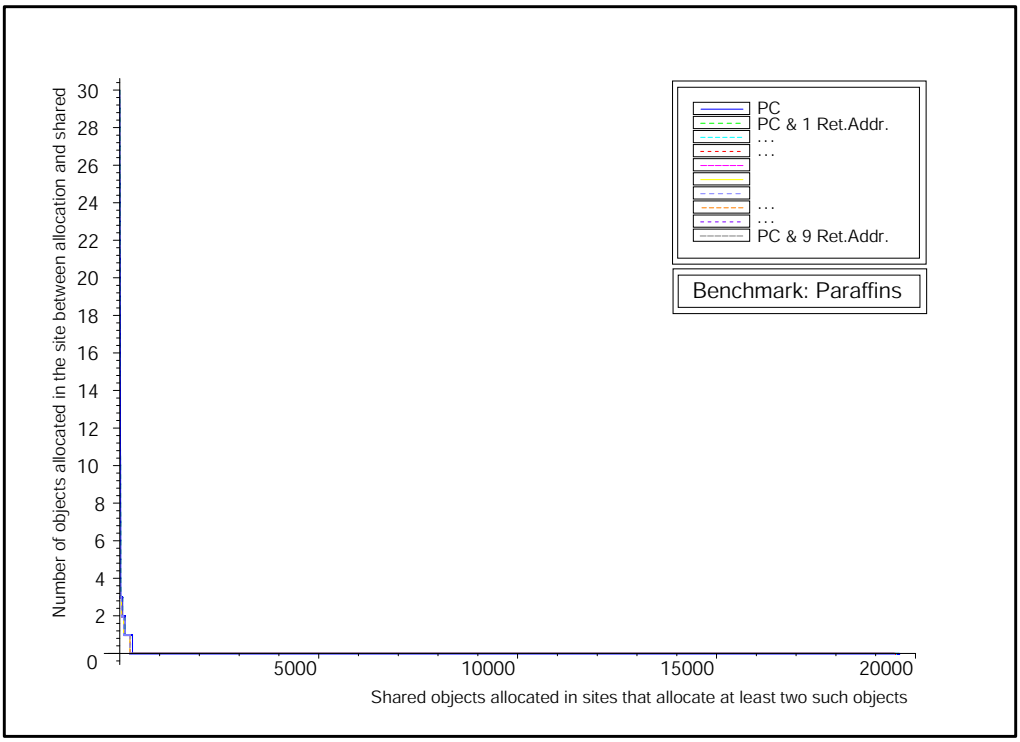


(b) Delay graph

Figure B.12.4: Benchmark: Pretzel



(a) Correlation graph



(b) Delay graph

Figure B.12.5: Benchmark: Paraffins

Nothing is so easy as to deceive one's self; for what we wish, we readily believe.
*— **Demosthenes**, 384 BC - 322 BC*

Appendix C

Examples

C.1 Pointer discovery in the registers

The following test program is compiled without optimisation.

```
int ping(int *);
int *pang(int,int*);
void pong(int);

void tinker(int a,int *b)
{
  pong(a+ping(pang(a,b)));
}
```

Non-optimised SPARC V8 code generated by GCC for the body:

```
  11 0004 F027A044            st      %i0, [%fp+68]
  12 0008 F227A048            st      %i1, [%fp+72]
  13 000c D007A044            ld      [%fp+68], %o0
  14 0010 D207A048            ld      [%fp+72], %o1
  15 0014 40000000            call    pang, 0
  16 0018 01000000            nop
  17 001c 82100008            mov     %o0, %g1
  18 0020 90100001            mov     %g1, %o0
  19 0024 40000000            call    ping, 0
  20 0028 01000000            nop
  21 002c 9A100008            mov     %o0, %o5
  22 0030 C207A044            ld      [%fp+68], %g1
  23 0034 82034001            add     %o5, %g1, %g1
  24 0038 90100001            mov     %g1, %o0
  25 003c 40000000            call    pong, 0
  26 0040 01000000            nop
```

Non-optimised SPARC V8 code generated by the customised GCC for the body, with mode annotations:

```
  16 0004 F027A044            st      %i0, [%fp+68]
  17                  #*TT*148 :%i0 #180
```

```
  18 0008 F227A048              st      %i1, [%fp+72]
  19                    #*TT*157 >%i1 ^184
  20 000c D007A044              ld      [%fp+68], %o0
  21                    #*TT*146 :180 #%o0
  22 0010 D207A048              ld      [%fp+72], %o1
  23                    #*TT*155 >184 ^%o1
  24 0014 40000000              call    pang, 0
  25                    #*TT*218C >%o1 :%o0 ^%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
  26 0018 01000000               nop
  27                    #*TT*800
  28 001c 82100008              mov     %o0, %g1
  29                    #*TT*151 >%o0 ^%g1
  30 0020 90100001              mov     %g1, %o0
  31                    #*TT*151 >%g1 ^%o0
  32 0024 40000000              call    ping, 0
  33                    #*TT*218C >%o0 #%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
  34 0028 01000000               nop
  35                    #*TT*800
  36 002c 9A100008              mov     %o0, %o5
  37                    #*TT*142 :%o0 #%o5
  38 0030 C207A044              ld      [%fp+68], %g1
  39                    #*TT*146 :180 #%g1
  40 0034 82034001              add     %o5, %g1, %g1
  41                    #*TT*219 #%g1 :%o5 :%g1
  42 0038 90100001              mov     %g1, %o0
  43                    #*TT*142 :%g1 #%o0
  44 003c 40000000              call    pong, 0
  45                    #*TT*213C :%o0 #%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
  46 0040 01000000               nop

Key:
> use ptr
^ def ptr
: use scal
# def scal

regs used: %o0, %o1, %i0, %i1, %g1, %o5 (%o0-%o5 clobbered by calls)
```

Initial tables, derived from the mode annotations.

```
initial tables, registers: %g1 %o0 %o1 %o5 %i0 %i1

           ptr           scal
        def   use      def   use    follow
0004  000000 000000  000000 000010  0008
0008  000000 000001  000000 000000  000c
000c  000000 000000  010000 000000  0010
0010  001000 000000  000000 000000  0014
0014  000000 000000  000000 000000  0018
0018  010000 001000  101100 010000  001c
001c  100000 010000  000000 000000  0020
0020  010000 100000  000000 000000  0024
0024  000000 000000  000000 000000  0028
0028  000000 010000  111100 000000  002c
002c  000000 000000  000100 010000  0030
0030  000000 000000  100000 000000  0034
0034  000000 000000  100000 100100  0038
0038  000000 000000  010000 100000  003c
003c  000000 000000  000000 000000  0040
0040  000000 000000  111100 010000  ---
```

Tables after the mode calculation:

```
Output, registers: %g1 %o0 %o1 %o5 %i0 %i1

     ptrs   scalars
0004 .....X ....X.
0008 .....X ......
000c ...... ......
0010 ...... .X....
0014 ..X... .X....
0018 ..X... .X....
001c .X.... ......
0020 X..... ......
0024 .X.... ......
```

```
0028 .X.... ......
002c ...... .X....
0030 ...... ...X..
0034 ...... X..X..
0038 ...... X.....
003c ...... .X....
0040 ...... .X....

-----
```

The resulting code with the pointer/scalar information. A flag set to ‘x’ means that the corresponding register is a pointer before that instruction.

```
Result, columns are: %g1 %o0 %o1 %o5 %i0 %i1

F027A044  0004 .....X  st      %i0, [%fp+68]
F227A048  0008 .....X  st      %i1, [%fp+72]
D007A044  000c ......  ld      [%fp+68], %o0
D207A048  0010 ......  ld      [%fp+72], %o1
40000000  0014 ..X...  call    pang, 0
01000000  0018 ..X...  nop
82100008  001c .X....  mov     %o0, %g1
90100001  0020 X.....  mov     %g1, %o0
40000000  0024 .X....  call    ping, 0
01000000  0028 .X....  nop
9A100008  002c ......  mov     %o0, %o5
C207A044  0030 ......  ld      [%fp+68], %g1
82034001  0034 ......  add     %o5, %g1, %g1
90100001  0038 ......  mov     %g1, %o0
40000000  003c ......  call    pong, 0
01000000  0040 ......  nop
```

C.2 Fully optimised code

The original program:

```
int ping(int *);
int *pang(int,int*);
int pong(int*,int);
void peng(int);
int *poing(int);

void tinker(int a,int *b)
{
  int i,j=0;

  for (i=ping(b);i<ping(poing(j));i++) {
   peng(a+ping(pang(a,b)));
   while (pong(poing(a),pong(b,1))) {
    if (j<91) {
     peng(a-1);
     j=pong(b,a+2) ? a*ping(b) : 0;
    } else {
     j=ping(pang(j,b));
     break;
    }
   }
  }
}
```

The fully optimised compiled code, with annotations:

```
        .file   "x.c"
        .global .umul
        .section        ".text"
        .align 4
        .global tinker
        .type   tinker, #function
        .proc   020
tinker:
#*TT*790S "tinker"
#*TT*955 16 112 2
#*TT*777 #%i0 ^%i1
#*TT*778
        save    %sp, -112, %sp
#*TT*029W
#*TT*223E
#*TT*223S
        mov     %i1, %o0
#*TT*151 >%i1 ^%o0
        call    ping, 0
#*TT*218C >%o0 #%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
        mov     0, %l1
#*TT*142 #%l1
        b       .LL2
#*TT*811U ".LL2"
        mov     %o0, %l2
#*TT*142 :%o0 #%l2
.LL14:
        call    pang, 0
#*TT*218C >%o1 :%o0 ^%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
        mov     %i0, %o0
#*TT*142 :%i0 #%o0
        call    ping, 0
#*TT*218C >%o0 #%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
         nop
#*TT*800
        call    peng, 0
#*TT*213C :%o0 #%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
        add     %i0, %o0, %o0
#*TT*219 #%o0 :%i0 :%o0
.LL15:
        call    poing, 0
#*TT*218C :%o0 ^%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
        mov     %i0, %o0
#*TT*142 :%i0 #%o0
        mov     1, %o1
```

```
#*TT*142 #%o1
         mov     %o0, %l0
#*TT*151 >%o0 ^%l0
         call    pong, 0
#*TT*218C :%o1 >%o0 #%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
         mov     %i1, %o0
#*TT*151 >%i1 ^%o0
         mov     %o0, %o1
#*TT*142 :%o0 #%o1
         call    pong, 0
#*TT*218C :%o1 >%o0 #%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
         mov     %l0, %o0
#*TT*151 >%l0 ^%o0
         mov     %o0, %o1
#*TT*142 :%o0 #%o1
         cmp     %o1, 0
#*TT*100 :%o1
         be      .LL7
#*TT*802B ".LL7"
         add     %i0, -1, %o0
#*TT*219 #%o0 :%i0
         cmp     %l1, 90
#*TT*100 :%l1
         bg      .LL9
#*TT*802B ".LL9"
         mov     %i1, %o1
#*TT*151 >%i1 ^%o1
         call    peng, 0
#*TT*213C :%o0 #%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
         mov     0, %l1
#*TT*142 #%l1
         add     %i0, 2, %o1
#*TT*219 #%o1 :%i0
         call    pong, 0
#*TT*218C :%o1 >%o0 #%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
         mov     %i1, %o0
#*TT*151 >%i1 ^%o0
         cmp     %o0, 0
#*TT*100 :%o0
         be      .LL15
#*TT*802B ".LL15"
         nop
#*TT*803
         call    ping, 0
#*TT*218C >%o0 #%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
         mov     %i1, %o0
#*TT*151 >%i1 ^%o0
         mov     %o0, %o1
#*TT*142 :%o0 #%o1
         call    .umul, 0
#*TT*218C :%o1 :%o0 #%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
         mov     %i0, %o0
#*TT*142 :%i0 #%o0
         b       .LL15
#*TT*811U ".LL15"
         mov     %o0, %l1
#*TT*142 :%o0 #%l1
.LL9:
         call    pang, 0
#*TT*218C >%o1 :%o0 ^%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
         mov     %l1, %o0
#*TT*142 :%l1 #%o0
         call    ping, 0
#*TT*218C >%o0 #%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
          nop
#*TT*800
         mov     %o0, %l1
#*TT*142 :%o0 #%l1
.LL7:
         add     %l2, 1, %l2
#*TT*219 #%l2 :%l2
.LL2:
         call    poing, 0
#*TT*218C :%o0 ^%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
         mov     %l1, %o0
#*TT*142 :%l1 #%o0
         call    ping, 0
#*TT*218C >%o0 #%o0 #%o1 #%o2 #%o3 #%o4 #%o5 #%o7 #%g1 #%g2 #%g2 #%g3 #%g4
          nop
#*TT*800
         cmp     %l2, %o0
```

```
#*TT*100 :%l2 :%o0
        bl      .LL14
#*TT*802B ".LL14"
        mov     %i1, %o1
#*TT*151 >%i1 ^%o1
#*TT*224E
#*TT*224S
        nop
#*TT*998
        ret
#*TT*404
        restore
#*TT*019W
#*TT*499E
        .size   tinker, .-tinker
        .ident  "GCC: (GNU) 3.3"
```

The resulting mode table, after processing. The part on the left has a ‘x’ if that register is used as a pointer, the part on the right if the register is used as a scalar.

```
##
## Final Masks (ptr/scal):
## 0004 ........:........:........:.X..... ........:........:........:X.......
## 0008 ........:X.......:........:.X..... ........:........:........:X.......
## 000c ........:X.......:........:.X..... ........:........:........:X.......
## 0010 ........:........:........:.X..... ........:X.......:.X......:X.......
## 0014 ........:........:........:.X..... ........:X.......:.X......:X.......
## 0018 ........:.X......:........:.X..... ........:........:.XX.....:X.......
## 001c ........:.X......:........:.X..... ........:........:.XX.....:X.......
## 0020 ........:X.......:........:.X..... ........:........:.XX.....:X.......
## 0024 ........:X.......:........:.X..... ........:........:.XX.....:X.......
## 0028 ........:........:........:.X..... ........:X.......:.XX.....:X.......
## 002c ........:........:........:.X..... ........:X.......:.XX.....:X.......
## 0030 ........:........:........:.X..... ........:........:.XX.....:X.......
## 0034 ........:........:........:.X..... ........:........:.XX.....:X.......
## 0038 ........:X.......:........:.X..... ........:........:.XX.....:X.......
## 003c ........:X.......:........:.X..... ........:.X......:.XX.....:X.......
## 0040 ........:........:X.......:.X..... ........:.X......:.XX.....:X.......
## 0044 ........:........:X.......:.X..... ........:.X......:.XX.....:X.......
## 0048 ........:........:X.......:.X..... ........:X.......:.XX.....:X.......
## 004c ........:........:X.......:.X..... ........:.X......:.XX.....:X.......
## 0050 ........:........:X.......:.X..... ........:.X......:.XX.....:X.......
## 0054 ........:........:........:.X..... ........:X.......:.XX.....:X.......
## 0058 ........:........:........:.X..... ........:.X......:.XX.....:X.......
## 005c ........:........:........:.X..... ........:........:.XX.....:X.......
## 0060 ........:........:........:.X..... ........:........:.XX.....:X.......
## 0064 ........:........:........:.X..... ........:X.......:.XX.....:X.......
## 0068 ........:........:........:.X..... ........:X.......:.XX.....:X.......
## 006c ........:........:........:.X..... ........:X.......:.XX.....:X.......
## 0070 ........:........:........:.X..... ........:X.......:..X.....:X.......
## 0074 ........:........:........:.X..... ........:X.......:..X.....:X.......
## 0078 ........:........:........:.X..... ........:........:.XX.....:X.......
## 007c ........:........:........:.X..... ........:.X......:.XX.....:X.......
## 0080 ........:........:........:.X..... ........:.X......:.XX.....:X.......
## 0084 ........:........:........:.X..... ........:X.......:.XX.....:X.......
## 0088 ........:........:........:.X..... ........:........:.XX.....:X.......
## 008c ........:........:........:.X..... ........:........:.XX.....:X.......
## 0090 ........:........:........:.X..... ........:........:..X.....:X.......
## 0094 ........:........:........:.X..... ........:........:..X.....:X.......
## 0098 ........:........:........:.X..... ........:X.......:..X.....:X.......
## 009c ........:........:........:.X..... ........:.X......:..X.....:X.......
## 00a0 ........:........:........:.X..... ........:.X......:..X.....:X.......
## 00a4 ........:........:........:.X..... ........:X.......:..X.....:X.......
## 00a8 ........:........:........:.X..... ........:X.......:..X.....:X.......
## 00ac ........:.X......:........:.X..... ........:........:.XX.....:X.......
## 00b0 ........:.X......:........:.X..... ........:........:.XX.....:X.......
## 00b4 ........:X.......:........:.X..... ........:........:..X.....:X.......
## 00b8 ........:X.......:........:.X..... ........:........:..X.....:X.......
## 00bc ........:........:........:.X..... ........:X.......:..X.....:X.......
## 00c0 ........:........:........:.X..... ........:........:.XX.....:X.......
## 00c4 ........:........:........:.X..... ........:........:.XX.....:X.......
## 00c8 ........:........:........:.X..... ........:........:.XX.....:X.......
## 00cc ........:X.......:........:.X..... ........:........:.XX.....:X.......
## 00d0 ........:X.......:........:.X..... ........:........:.XX.....:X.......
## 00d4 ........:........:........:.X..... ........:X.......:.XX.....:X.......
## 00d8 ........:........:........:.X..... ........:........:.XX.....:X.......
## 00dc ........:........:........:.X..... ........:........:.XX.....:X.......
##
```

The mode table, describing the registers used as pointers, after compression:

```
###                     --Registers used only as scalars
        .long   0xff3f7fbf
###                     --Registers used only as pointers
        .long   0x00000040
###                     --Table columns contain:
### Reg/offs: %o0
### Reg/offs: %o1
### Reg/offs: %l0
###                     --Header completed.
###                     --Compressed table:
        .long   0xcc343709
        .long   0x58700370
        .long   0xc3707bb7
        .long   0x0c000000
###                     --Table done.
```

C.3 Side-by-side comparison

This example shows a side-by-side comparison between the original GCC 3.3.1 compiler for the SPARC architecture and the customised compiler. The source file used for the test is appended at the end of the example. The code on the left is the original one, and the code on the right is the optimised code with the additional PC maps.

In this example the optimised code generated by the customised compiler is mostly indistinguishable from the original code. The additional PC maps appear on the right, immediately after each compiled routine.

```
        .file   "test.c"
        .global _TT_layout_node
        .section        ".data"
        .align 4
        .type   _TT_layout_node, #object
        .size   _TT_layout_node, 12
_TT_layout_node:
        .byte   0
        .byte   0
        .byte   0
        .byte   0
        .byte   0
        .byte   0
        .byte   0
        .byte   0
        .byte   4
        .byte   128
        .byte   0
        .byte   0
        .byte   0
        .section        ".text"
        .align 4
        .global add
        .type   add, #function
        .proc   020
add:
        !#PROLOGUE# 0
        save    %sp, -112, %sp
        !#PROLOGUE# 1
        sethi   %hi(_TT_layout_node), %g1
        or      %g1, %lo(_TT_layout_node), %o1
        call    trackAllok, 0
        mov     12, %o0
        st      %g0, [%o0]
        st      %g0, [%o0+4]
        ld      [%i0], %o1
        st      %i1, [%o0+8]
        mov     %o1, %o5
        cmp     %o1, 0
        bne     .LL13
        mov     1, %o4
        b       .LL1
        st      %o0, [%i0]
.LL13:
        ld      [%o5+8], %o1
.LL17:
        cmp     %o1, %i1
        bleu,a  .LL7
        ld      [%o5], %g1
        ld      [%o5+4], %g1
        cmp     %g1, 0
        be,a    .LL1
        st      %o0, [%o5+4]
        mov     %g1, %o5
.LL4:
        cmp     %o4, 0
        bne,a   .LL17
        ld      [%o5+8], %o1
        b,a     .LL1
.LL7:
        cmp     %g1, 0
        bne,a   .LL4
```

```
        .file   "test.c"
        .global _TT_layout_node
        .section        ".data"
        .align 4
        .type   _TT_layout_node, #object
        .size   _TT_layout_node, 12
_TT_layout_node:
        .byte   0
        .byte   0
        .byte   0
        .byte   0
        .byte   0
        .byte   0
        .byte   0
        .byte   0
        .byte   4
        .byte   128
        .byte   0
        .byte   0
        .byte   0
        .section        ".text"
        .align 4
        .global add
        .type   add, #function
        .proc   020
add:
<
        save    %sp, -112, %sp
<
        sethi   %hi(_TT_layout_node), %g1
        or      %g1, %lo(_TT_layout_node), %o1
        call    trackAllok, 0
        mov     12, %o0
        st      %g0, [%o0]
        st      %g0, [%o0+4]
        ld      [%i0], %o1
        st      %i1, [%o0+8]
        mov     %o1, %o5
        cmp     %o1, 0
        bne     .LL13
        mov     1, %o4
        b       .LL1
        st      %o0, [%i0]
.LL13:
        ld      [%o5+8], %o1
.LL17:
        cmp     %o1, %i1
        bleu,a  .LL7
        ld      [%o5], %g1
        ld      [%o5+4], %g1
        cmp     %g1, 0
        be,a    .LL1
        st      %o0, [%o5+4]
        mov     %g1, %o5
.LL4:
        cmp     %o4, 0
        bne,a   .LL17
        ld      [%o5+8], %o1
        b,a     .LL1
.LL7:
        cmp     %g1, 0
        bne,a   .LL4
```

```
        mov     %g1, %o5
        st      %o0, [%o5]
        b       .LL4
        mov     0, %o4
.LL1:

        ret
        restore
        .size   add, .-add

        .section        .rodata.str1.8,"aMS",@progbits,1
        .align 8
.LLC0:
        .asciz  "%d "
        .section        ".text"
        .align 4
        .global printTree
        .type   printTree, #function
        .proc   020
printTree:
        !#PROLOGUE# 0
        save    %sp, -112, %sp
        !#PROLOGUE# 1
        orcc    %i0, 0, %l0
        be      .LL18
        nop
        call    printTree, 0
        ld      [%l0+4], %o0
        ld      [%l0+8], %o1
        sethi   %hi(.LLC0), %g1
        call    printf, 0
        or      %g1, %lo(.LLC0), %o0
        call    printTree, 0
        ld      [%l0], %o0
```

```
          mov     %g1, %o5
          st      %o0, [%o5]
          b       .LL4
          mov     0, %o4
 .LL1:
>         nop
          ret
          restore
          .size   add, .-add
>         .global __TT__add_end
> __TT__add_end:
>         .section        ".rodata"
>         .align 4
>         .type   __TT__add_regTable,#object
>         .size   __TT__add_regTable,__TT__add_regTable-__TT__a
>         .global __TT__add_regTable
> __TT__add_regTable:
> ###                     --Save and restore offsets for regist
> ###                       or add/sub offsets for stack pointe
>         .long   0x00000000
>         .long   0x00000090
> ###                     --Start of body and Start of epilogue
>         .long   0x00000004
>         .long   0x00000088
> ###                     --Flags: saveRestoreUsed
> ###                              retIsPtr
> ###                              spMoved
>         .long   0xa0000000
> ###                     --Frame size:
>         .long   0x00000070
> ###                     --Outgoing params area size:
>         .long   0x00000060
> ###                     --Number of stack slots ever used as
>         .long   0x00000000
> ###                     --Used offsets:
> ###                     --Registers used only as scalars
>         .long   0xbf3bff7f
> ###                     --Registers used only as pointers
>         .long   0x00000000
> ###                     --Table columns contain:
> ### Reg/offs: %g1
> ### Reg/offs: %o0
> ### Reg/offs: %o1
> ### Reg/offs: %o5
> ### Reg/offs: %i0
> ###                     --Header completed.
> ###                     --Compressed table:
>         .long   0x97035a17
>         .long   0x00e193f1
>         .long   0x83f0b5cf
>         .long   0x7daae0fc
>         .long   0x12000000
> ###                     --Table done.
>         .global __TT__add_regTable_end
> __TT__add_regTable_end:
>
>         .section        ".text"
          .section        .rodata.str1.8,"aMS",@progbits,1
          .align 8
 .LLC0:
          .asciz  "%d "
          .section        ".text"
          .align 4
          .global printTree
          .type   printTree, #function
          .proc   020
 printTree:
<
          save    %sp, -112, %sp
|         cmp     %i0, 0
<
          be      .LL18
          nop
          call    printTree, 0
|         ld      [%i0+4], %o0
|         ld      [%i0+8], %o1
          sethi   %hi(.LLC0), %g1
          call    printf, 0
          or      %g1, %lo(.LLC0), %o0
          call    printTree, 0
|         ld      [%i0], %o0
```

```
        call    trackRelease, 0
         restore
.LL18:
        nop
        ret
        restore
        .size   printTree, .-printTree
```

```
        call    trackRelease, 0
|       mov     %i0, %o0
 .LL18:
        nop
        ret
        restore
        .size   printTree, .-printTree
>       .global __TT__printTree_end
> __TT__printTree_end:
>       .section        ".rodata"
>       .align 4
>       .type   __TT__printTree_regTable,#object
>       .size   __TT__printTree_regTable,__TT__printTree_regT
>       .global __TT__printTree_regTable
> __TT__printTree_regTable:
> ###                   --Save and restore offsets for regist
> ###                     or add/sub offsets for stack pointe
>       .long   0x00000000
>       .long   0x00000040
> ###                   --Start of body and Start of epilogue
>       .long   0x00000004
>       .long   0x00000038
> ###                   --Flags: saveRestoreUsed
> ###                            retIsPtr
> ###                            spMoved
>       .long   0xa0000000
> ###                   --Frame size:
>       .long   0x00000070
> ###                   --Outgoing params area size:
>       .long   0x00000060
> ###                   --Number of stack slots ever used as
>       .long   0x00000000
> ###                   --Used offsets:
> ###                   --Registers used only as scalars
>       .long   0xbfffff7f
> ###                   --Registers used only as pointers
>       .long   0x00000080
> ###                   --Table columns contain:
> ### Reg/offs: %g1
> ###                   --Header completed.
> ###                   --Compressed table:
>       .long   0x00c00000
> ###                   --Table done.
>       .global __TT__printTree_regTable_end
> __TT__printTree_regTable_end:
>
>       .section        ".text"
```

```
        .align 4
        .global printInt
        .type   printInt, #function
        .proc   020
printInt:
        !#PROLOGUE# 0
        !#PROLOGUE# 1
        mov     %o0, %o1
        sethi   %hi(.LLC0), %g1
        or      %g1, %lo(.LLC0), %o0
        or      %o7, %g0, %g1
        call    printf, 0
         or     %g1, %g0, %o7
        nop


        .size   printInt, .-printInt
```

```
        .align 4
        .global printInt
        .type   printInt, #function
        .proc   020
 printInt:
|       save    %sp, -112, %sp
<
<
        sethi   %hi(.LLC0), %g1
|       mov     %i0, %o1
<
        call    printf, 0
|       or      %g1, %lo(.LLC0), %o0
        nop
>       ret
>       restore
        .size   printInt, .-printInt
>       .global __TT__printInt_end
> __TT__printInt_end:
>       .section        ".rodata"
>       .align 4
>       .type   __TT__printInt_regTable,#object
>       .size   __TT__printInt_regTable,__TT__printInt_regTab
>       .global __TT__printInt_regTable
> __TT__printInt_regTable:
> ###                   --Save and restore offsets for regist
> ###                     or add/sub offsets for stack pointe
>       .long   0x00000000
>       .long   0x0000001c
> ###                   --Start of body and Start of epilogue
>       .long   0x00000004
>       .long   0x00000014
> ###                   --Flags: saveRestoreUsed
> ###                            retIsPtr
```

```
> ###                             spMoved
>         .long   0xa0000000
> ###                     --Frame size:
>         .long   0x00000070
> ###                     --Outgoing params area size:
>         .long   0x00000060
> ###                     --Number of stack slots ever used as
>         .long   0x00000000
> ###                     --Used offsets:
> ###                     --Registers used only as scalars
>         .long   0xbfffffff
> ###                     --Registers used only as pointers
>         .long   0x00000000
> ###                     --Table columns contain:
> ### Reg/offs: %g1
> ###                     --Header completed.
> ###                     --Compressed table:
>         .long   0x38000000
> ###                     --Table done.
>         .global __TT__printInt_regTable_end
> __TT__printInt_regTable_end:
>
>         .section        ".text"
          .align 4
          .global newNode
          .type   newNode, #function
          .proc   0110
 newNode:
<
          save    %sp, -112, %sp
<
          sethi   %hi(_TT_layout_node), %g1
          mov     12, %o0
          call    trackAllok, 0
          or      %g1, %lo(_TT_layout_node), %o1
          st      %i0, [%o0+8]
          st      %g0, [%o0]
          st      %g0, [%o0+4]
>         mov     %o0, %i0
>         nop
          ret
|         restore
          .size   newNode, .-newNode
>         .global __TT__newNode_end
> __TT__newNode_end:
>         .section        ".rodata"
>         .align 4
>         .type   __TT__newNode_regTable,#object
>         .size   __TT__newNode_regTable,__TT__newNode_regTable
>         .global __TT__newNode_regTable
> __TT__newNode_regTable:
> ###                     --Save and restore offsets for regist
> ###                       or add/sub offsets for stack pointe
>         .long   0x00000000
>         .long   0x0000002c
> ###                     --Start of body and Start of epilogue
>         .long   0x00000004
>         .long   0x00000024
> ###                     --Flags: saveRestoreUsed
> ###                              retIsPtr
> ###                              spMoved
>         .long   0xe0000000
> ###                     --Frame size:
>         .long   0x00000070
> ###                     --Outgoing params area size:
>         .long   0x00000060
> ###                     --Number of stack slots ever used as
>         .long   0x00000000
> ###                     --Used offsets:
> ###                     --Registers used only as scalars
>         .long   0xbf7fffff
> ###                     --Registers used only as pointers
>         .long   0x00000000
> ###                     --Table columns contain:
> ### Reg/offs: %g1
> ### Reg/offs: %o0
> ###                     --Header completed.
> ###                     --Compressed table:
>         .long   0x38078000
> ###                     --Table done.
>         .global __TT__newNode_regTable_end
```

```
        .section        .rodata.str1.8
        .align 8
.LLC1:
        .asciz  "Filling the tree (in C!)..."
        .align 8
.LLC2:
        .asciz  "Emptying the tree (in C!)..."
        .align 8
.LLC3:
        .asciz  "\n...emptied."
        .align 8
.LLC4:
        .asciz  "Filling the tree (in Ada!)..."
        .align 8
.LLC5:
        .asciz  "Emptying the tree (in Ada!)..."
        .section        ".text"
        .align 4
        .global trackMain
        .type   trackMain, #function
        .proc   04
trackMain:
        !#PROLOGUE# 0
        save    %sp, -120, %sp
        !#PROLOGUE# 1
        sethi   %hi(4096), %o0
        mov     0, %l3
        mov     %o0, %i3
        sethi   %hi(.LLC1), %l6
        sethi   %hi(.LLC2), %l7
        sethi   %hi(.LLC3), %i5
        sethi   %hi(.LLC4), %i4
        sethi   %hi(_TT_layout_node), %l2
        sethi   %hi(.LLC5), %l5
        or      %o0, 903, %l4
        or      %l6, %lo(.LLC1), %o0
.LL49:
        call    puts, 0
        mov     0, %i0
        st      %g0, [%fp-20]
.LL32:
        call    rand, 0
        add     %i0, 1, %i0
        mov     %o0, %o1
        call    add, 0
        add     %fp, -20, %o0
        cmp     %i0, %l4
        bleu    .LL32
        nop
        call    puts, 0
        or      %l7, %lo(.LLC2), %o0
        ld      [%fp-20], %l0
        cmp     %l0, 0
        be      .LL34
        nop
        call    printTree, 0
        ld      [%l0+4], %o0
        ld      [%l0+8], %o1
        sethi   %hi(.LLC0), %g1
        call    printf, 0
        or      %g1, %lo(.LLC0), %o0
        call    printTree, 0
        ld      [%l0], %o0
        call    trackRelease, 0
        mov     %l0, %o0
.LL34:
        call    puts, 0
        or      %i5, %lo(.LLC3), %o0
        call    puts, 0
        or      %i4, %lo(.LLC4), %o0
        call    rand, 0
        st      %g0, [%fp-20]
        or      %l2, %lo(_TT_layout_node), %o1
        mov     %o0, %l1
        call    trackAllok, 0
        mov     12, %o0
        st      %g0, [%o0]
        st      %g0, [%o0+4]
> __TT__newNode_regTable_end:
>
>       .section        ".text"
        .section        .rodata.str1.8
        .align 8
 .LLC1:
        .asciz  "Filling the tree (in C!)..."
        .align 8
 .LLC2:
        .asciz  "Emptying the tree (in C!)..."
        .align 8
 .LLC3:
        .asciz  "\n...emptied."
        .align 8
 .LLC4:
        .asciz  "Filling the tree (in Ada!)..."
        .align 8
 .LLC5:
        .asciz  "Emptying the tree (in Ada!)..."
        .section        ".text"
        .align 4
        .global trackMain
        .type   trackMain, #function
        .proc   04
 trackMain:
<
        save    %sp, -120, %sp
<
        sethi   %hi(4096), %o0
        mov     0, %l3
        mov     %o0, %i3
|       sethi   %hi(.LLC1), %l5
|       sethi   %hi(.LLC2), %l6
|       sethi   %hi(.LLC3), %l7
|       sethi   %hi(.LLC4), %i5
        sethi   %hi(_TT_layout_node), %l2
|       sethi   %hi(.LLC5), %i4
        or      %o0, 903, %l4
|       or      %l5, %lo(.LLC1), %o0
 .LL49:
        call    puts, 0
        mov     0, %i0
        st      %g0, [%fp-20]
 .LL32:
        call    rand, 0
        add     %i0, 1, %i0
        mov     %o0, %o1
        call    add, 0
        add     %fp, -20, %o0
        cmp     %i0, %l4
        bleu    .LL32
        nop
        call    puts, 0
|       or      %l6, %lo(.LLC2), %o0
        ld      [%fp-20], %l0
        cmp     %l0, 0
        be      .LL34
        nop
        call    printTree, 0
        ld      [%l0+4], %o0
        ld      [%l0+8], %o1
        sethi   %hi(.LLC0), %g1
        call    printf, 0
        or      %g1, %lo(.LLC0), %o0
        call    printTree, 0
        ld      [%l0], %o0
        call    trackRelease, 0
        mov     %l0, %o0
 .LL34:
        call    puts, 0
|       or      %l7, %lo(.LLC3), %o0
        call    puts, 0
|       or      %i5, %lo(.LLC4), %o0
        call    rand, 0
        st      %g0, [%fp-20]
        or      %l2, %lo(_TT_layout_node), %o1
        mov     %o0, %l1
        call    trackAllok, 0
        mov     12, %o0
        st      %g0, [%o0]
        st      %g0, [%o0+4]
```

```
        st      %l1, [%o0+8]
        st      %o0, [%fp-20]
        mov     1, %i0
        or      %i3, 903, %l1
.LL41:
        call    rand, 0
        add     %i0, 1, %i0
        or      %l2, %lo(_TT_layout_node), %o1
        mov     %o0, %l0
        call    trackAllok, 0
        mov     12, %o0
        mov     %o0, %o1
        st      %l0, [%o0+8]
        st      %g0, [%o0]
        st      %g0, [%o0+4]
        call    integral__add, 0
        ld      [%fp-20], %o0
        cmp     %i0, %l1
        bleu    .LL41
        nop
        call    puts, 0
        or      %l5, %lo(.LLC5), %o0
        ld      [%fp-20], %o0
        call    integral__scan, 0
        add     %l3, 1, %l3
        call    puts, 0
        or      %i5, %lo(.LLC3), %o0
        cmp     %l3, 99
        bleu    .LL49
        or      %l6, %lo(.LLC1), %o0
        mov     10, %o0
        call    putchar, 0
        mov     0, %i0

        ret
        restore
        .size   trackMain, .-trackMain
```

```
        st      %l1, [%o0+8]
        st      %o0, [%fp-20]
        mov     1, %i0
        or      %i3, 903, %l1
.LL41:
        call    rand, 0
        add     %i0, 1, %i0
        or      %l2, %lo(_TT_layout_node), %o1
        mov     %o0, %l0
        call    trackAllok, 0
        mov     12, %o0
        mov     %o0, %o1
        st      %l0, [%o0+8]
        st      %g0, [%o0]
        st      %g0, [%o0+4]
        call    integral__add, 0
        ld      [%fp-20], %o0
        cmp     %i0, %l1
        bleu    .LL41
        nop
        call    puts, 0
|       or      %i4, %lo(.LLC5), %o0
        ld      [%fp-20], %o0
        call    integral__scan, 0
        add     %l3, 1, %l3
        call    puts, 0
|       or      %l7, %lo(.LLC3), %o0
        cmp     %l3, 99
        bleu    .LL49
|       or      %l5, %lo(.LLC1), %o0
        mov     10, %o0
        call    putchar, 0
        mov     0, %i0
>       nop
        ret
        restore
        .size   trackMain, .-trackMain
>       .global __TT__trackMain_end
> __TT__trackMain_end:
>       .section        ".rodata"
>       .align 4
>       .type   __TT__trackMain_regTable,#object
>       .size   __TT__trackMain_regTable,__TT__trackMain_regT
>       .global __TT__trackMain_regTable
> __TT__trackMain_regTable:
> ###                   --Save and restore offsets for regist
> ###                     or add/sub offsets for stack pointe
>       .long   0x00000000
>       .long   0x00000154
> ###                   --Start of body and Start of epilogue
>       .long   0x00000004
>       .long   0x0000014c
> ###                   --Flags: saveRestoreUsed
> ###                            retIsPtr
> ###                            spMoved
>       .long   0xa0000000
> ###                   --Frame size:
>       .long   0x00000078
> ###                   --Outgoing params area size:
>       .long   0x00000060
> ###                   --Number of stack slots ever used as
>       .long   0x00000001
> ###                   --Used offsets:
>       .long   0x00000064
> ###                   --Registers used only as scalars
>       .long   0xbf3f58f3
> ###                   --Registers used only as pointers
>       .long   0x00000000
>       .long   0x00000000
> ###                   --Table columns contain:
> ### Reg/offs: %g1
> ### Reg/offs: %o0
> ### Reg/offs: %o1
> ### Reg/offs: %l0
> ### Reg/offs: %l2
> ### Reg/offs: %l5
> ### Reg/offs: %l6
> ### Reg/offs: %l7
> ### Reg/offs: %i4
> ### Reg/offs: %i5
> ### Reg/offs: 100
```

```
> ###                   --Header completed.
> ###                   --Compressed table:
>         .long   0xf0b37183
>         .long   0x70c5e107
>         .long   0xc00dc557
>         .long   0x041cedc3
>         .long   0xd8271567
>         .long   0xd00e8300
>         .long   0x682001a0
>         .long   0x70167bc0
>         .long   0x19ff03e0
>         .long   0x5c2385f0
>         .long   0x00000000
> ###                   --Table done.
>         .global __TT__trackMain_regTable_end
> __TT__trackMain_regTable_end:
>
>         .section        ".text"
        .ident  "GCC: (GNU) 3.3.1"                      .ident  "GCC: (GNU) 3.3.1"
>
```

The following is the source file from which the above assembly code was generated.

```
//
// $RCSfile: test.c,v $
//
// $Author: toni $
// $Revision: 1.1 $
// $Date: 2003/04/10 23:39:22 $
//
//---------------------------------------------------------------------
#include "alpha.h"
// last, range, offs
lEntry _TT_layout_node[]={{0,0,0},{0,0,4},{1,0,0}};

void add(node **tree,uint32 x)
{
  node *n=trackAlloc(node);
  n->data=x;
  n->r=NULL;
  n->l=NULL;

  if (!*tree) {
   *tree=n;
  } else {
   node *target=*tree;
   uint32 workLeft=1;

  while (workLeft) {
    if (target->data > x) {
     if (target->l)
      target=target->l;
     else {
      target->l=n;
      workLeft=0;
     }
    } else {
     if (target->r)
      target=target->r;
     else {
      target->r=n;
      workLeft=0;
     }
    }
   }
  }
}

void printTree(node *n)
{
  if (n) {
   printTree(n->l);
   printf("%d ",n->data);
   printTree(n->r);
```

```
   trackRelease(n);
  }
}

//---- Ada interface
void printInt(int i)
{
   printf("%d ",i);
}

node *newNode(uint32 x)
{
  node *n=trackAlloc(node);
  n->data=x;
  n->r=NULL;
  n->l=NULL;
  return n;
}

//----

trackMain()
{
  node *root;
  uint32 i,j;
  for (j=0;j<100;j++) {

   printf("Filling the tree (in C!)...\n");
   root=NULL;
   for (i=0;i<5000;i++)
    add(&root,(uint32)rand());
   printf("Emptying the tree (in C!)...\n");
   printTree(root);
   printf("\n...emptied.\n");

   printf("Filling the tree (in Ada!)...\n");
   root=NULL;
   root=newNode((uint32)rand());
   for (i=1;i<5000;i++)
    integral__add(root,newNode((uint32)rand()));
   printf("Emptying the tree (in Ada!)...\n");
   integral__scan(root);
   printf("\n...emptied.\n");

  }
  printf("\n");
  return 0;
}
```

C.4 Tables from multiple languages

C.4.1 Java

```
public class Tree {

Tree r,l;
int data;

public static void add(Tree root,Tree in)
{
   boolean workLeft=true;
   Tree target=root;
   int x=in.data;

   while (workLeft) {
    if (target.data > x) {
     if (target.l!=null)
      target=target.l;
     else {
      target.l=in;
      workLeft=false;
     }
    } else {
     if (target.r!=null)
      target=target.r;
     else {
      target.r=in;
      workLeft=false;
     }
    }
   }
}
```

```
        .global __TT___ZN4Tree3addEPS_S0__regTable
__TT___ZN4Tree3addEPS_S0__regTable:
###                     --Save and restore offsets for register window shift
###                       or add/sub offsets for stack pointer adjustment
        .long   0x00000000
        .long   0x0000006c
###                     --Start of body and Start of epilogue offsets
        .long   0x00000004
        .long   0x00000064
###                     --Flags: saveRestoreUsed
###                              retIsPtr
###                              spMoved
        .long   0xa0000000
###                     --Frame size:
        .long   0x00000070
###                     --Outgoing params area size:
        .long   0x00000060
###                     --Number of stack slots ever used as ptrs:
        .long   0x00000000
###                     --Used offsets:
###                     --Registers used only as scalars
        .long   0xbfffff3f
###                     --Registers used only as pointers
        .long   0x00000040
###                     --Table columns contain:
### Reg/offs: %g1
### Reg/offs: %i0
###                     --Header completed.
###                     --Compressed table:
        .long   0x30038e7f
        .long   0xff7d8000
###                     -- Table done.
###                     --   49 bits used in compressed table
###                     --    plus two times 32 bits used in the header
###                     --   768 bits for the full table.
###                     --   compression squeezed down to: 14.7135% of original
###                     --    and 6.38021% excluding the header.
###                     --   Compressed table (excluding header), represented
###                     --    using whole 32-bit words, was
###                     --    8.33333% of the size of the code in the body,
###                     --    but only 6.38021% counting the bits really used.
        .global __TT___ZN4Tree3addEPS_S0__regTable_end
__TT___ZN4Tree3addEPS_S0__regTable_end:
##
## Final Masks (ptr/scal):
```

```
## 0004 ........:........:........:XX...... ........:........:........:........
## 0008 .X......:........:........:XX...... ........:........:........:........
## 000c .X......:........:........:XX...... ........:........:........:........
## 0010 ........:........:........:XX...... ........:........:........:........
## 0014 ........:........:........:XX...... ........:....X...:........:........
## 0018 ........:........:........:XX...... ........:....XX..:........:........
## 001c ........:........:........:XX...... ........:....XX..:........:........
## 0020 ........:........:........:XX...... ........:....XX..:........:........
## 0024 ........:........:........:XX...... ........:....XX..:........:........
## 0028 ........:........:........:XX...... ........:X...XX..:........:........
## 002c ........:........:........:XX...... ........:....XX..:........:........
## 0030 ........:........:........:XX...... ........:....XX..:........:........
## 0034 ........:........:........:XX...... ........:....XX..:........:........
## 0038 .X......:........:........:XX...... ........:....XX..:........:........
## 003c .X......:........:........:XX...... ........:....XX..:........:........
## 0040 .X......:........:........:.X...... ........:....XX..:........:........
## 0044 ........:........:........:XX...... ........:....X...:........:........
## 0048 ........:........:........:XX...... ........:....X...:........:........
## 004c ........:........:........:XX...... ........:....X...:........:........
## 0050 .X......:........:........:XX...... ........:....XX..:........:........
## 0054 .X......:........:........:XX...... ........:....XX..:........:........
## 0058 .X......:........:........:.X...... ........:....XX..:........:........
## 005c ........:........:........:XX...... ........:....X...:........:........
## 0060 ........:........:........:XX...... ........:....X...:........:........
##
```

C.4.2 C language

```
int pip(int a,int ta,int tb,int tc,int td,int te,int tf,int tg,int th,int ti,
                int tj,int tk,int tl,int tm,int tn,int to,int tp,int tq,int tr,
                int ts,int tt,int tu,int tv,int tw,int tx,int ty,int tz)
{
  int b=a>>1;
  int c=a>>(a+b);
  int d=a>>((b>>c));
  int e=a>>((b>>d)+(c>>d));
  int f=a>>((b>>e)+(c>>e)+(d>>e));
  int g=a>>((b>>f)+(c>>f)+(d>>f)+(e>>f));
  int h=a>>((b>>g)+(c>>g)+(d>>g)+(e>>g)+(f>>g));
  int i=a>>((b>>h)+(c>>h)+(d>>h)+(e>>h)+(f>>h)+(g>>h));
  int j=a>>((b>>i)+(c>>i)+(d>>i)+(e>>i)+(f>>i)+(g>>i)+(h>>i));
  int k=a>>((b>>j)+(c>>j)+(d>>j)+(e>>j)+(f>>j)+(g>>j)+(h>>j)+(i>>j));
  int l=a>>((b>>k)+(c>>k)+(d>>k)+(e>>k)+(f>>k)+(g>>k)+(h>>k)+(i>>k)+(j>>k));
  int m=a>>((b>>l)+(c>>l)+(d>>l)+(e>>l)+(f>>l)+(g>>l)+(h>>l)+(i>>l)+(j>>l)+(k>>l));
  int n=a>>((b>>m)+(c>>m)+(d>>m)+(e>>m)+(f>>m)+(g>>m)+(h>>m)+(i>>m)+(j>>m)+(k>>m)+(l>>m));
  int o=a>>((b>>n)+(c>>n)+(d>>n)+(e>>n)+(f>>n)+(g>>n)+(h>>n)+(i>>n)+(j>>n)+(k>>n)+(l>>n)+(m>>n));
  int p=a>>((b>>o)+(c>>o)+(d>>o)+(e>>o)+(f>>o)+(g>>o)+(h>>o)+(i>>o)+(j>>o)+(k>>o)+(l>>o)+(m>>o)+(n>>o));
  int q=a>>((b>>p)+(c>>p)+(d>>p)+(e>>p)+(f>>p)+(g>>p)+(h>>p)+(i>>p)+(j>>p)+(k>>p)+(l>>p)+(m>>p)+(n>>p)+(o>>p));
  int r=a>>((b>>q)+(c>>q)+(d>>q)+(e>>q)+(f>>q)+(g>>q)+(h>>q)+(i>>q)+(j>>q)+(k>>q)+(l>>q)+(m>>q)+(n>>q)+(o>>q)+(p>>q));
  int s=a>>((b>>r)+(c>>r)+(d>>r)+(e>>r)+(f>>r)+(g>>r)+(h>>r)+(i>>r)+(j>>r)+(k>>r)+(l>>r)+(m>>r)+(n>>r)+(o>>r)+(p>>r)+(q>>r));
  int t=a>>((b>>s)+(c>>s)+(d>>s)+(e>>s)+(f>>s)+(g>>s)+(h>>s)+(i>>s)+(j>>s)+(k>>s)+(l>>s)+(m>>s)+(n>>s)+(o>>s)+(p>>s)+(q>>s)+
    (r>>s));
  int u=a>>((b>>t)+(c>>t)+(d>>t)+(e>>t)+(f>>t)+(g>>t)+(h>>t)+(i>>t)+(j>>t)+(k>>t)+(l>>t)+(m>>t)+(n>>t)+(o>>t)+(p>>t)+(q>>t)+
    (r>>t)+(s>>t));
  int v=a>>((b>>u)+(c>>u)+(d>>u)+(e>>u)+(f>>u)+(g>>u)+(h>>u)+(i>>u)+(j>>u)+(k>>u)+(l>>u)+(m>>u)+(n>>u)+(o>>u)+(p>>u)+(q>>u)+
    (r>>u)+(s>>u)+(t>>u));
  int w=a>>((b>>v)+(c>>v)+(d>>v)+(e>>v)+(f>>v)+(g>>v)+(h>>v)+(i>>v)+(j>>v)+(k>>v)+(l>>v)+(m>>v)+(n>>v)+(o>>v)+(p>>v)+(q>>v)+
    (r>>v)+(s>>v)+(t>>v)+(u>>v));
  int x=a>>((b>>w)+(c>>w)+(d>>w)+(e>>w)+(f>>w)+(g>>w)+(h>>w)+(i>>w)+(j>>w)+(k>>w)+(l>>w)+(m>>w)+(n>>w)+(o>>w)+(p>>w)+(q>>w)+
    (r>>w)+(s>>w)+(t>>w)+(u>>w)+(v>>w));
  int y=a>>((b>>x)+(c>>x)+(d>>x)+(e>>x)+(f>>x)+(g>>x)+(h>>x)+(i>>x)+(j>>x)+(k>>x)+(l>>x)+(m>>x)+(n>>x)+(o>>x)+(p>>x)+(q>>x)+
    (r>>x)+(s>>x)+(t>>x)+(u>>x)+(v>>x)+(w>>x));
  int z=a>>((b>>y)+(c>>y)+(d>>y)+(e>>y)+(f>>y)+(g>>y)+(h>>y)+(i>>y)+(j>>y)+(k>>y)+(l>>y)+(m>>y)+(n>>y)+(o>>y)+(p>>y)+(q>>y)+
    (r>>y)+(s>>y)+(t>>y)+(u>>y)+(v>>y)+(w>>y)+(x>>y));
  return z+ta+tb+tc+td+te+tf+tg+th+ti+tj+tk+tl+tm+tn+to+tp+tq+tr+ts+tt+tu+tv+tw+tx+ty+tz;
}
```

```
        .global __TT__pip_regTable
__TT__pip_regTable:
###                     --Save and restore offsets for register window shift
###                       or add/sub offsets for stack pointer adjustment
        .long   0x00000000
        .long   0x00000c64
###                     --Start of body and Start of epilogue offsets
        .long   0x00000004
        .long   0x00000c5c
```

```
###                     --Flags: saveRestoreUsed
###                              retIsPtr
###                              spMoved
        .long   0xa0000000
###                     --Frame size:
        .long   0x00000098
###                     --Outgoing params area size:
        .long   0x00000060
###                     --Number of stack slots ever used as ptrs:
        .long   0x00000000
###                     --Used offsets:
###                     --Registers used only as scalars
        .long   0xffffffff
###                     --Registers used only as pointers
        .long   0x00000000
###                     --Table columns contain:
###                     --Header completed.
###                     --Compressed table:
###                     -- Table done.
###                     --   0 bits used in compressed table
###                     --    plus two times 32 bits used in the header
###                     --   25280 bits for the full table.
###                     --   compression squeezed down to: 0.253165% of original
###                     --    and 0% excluding the header.
###                     --   Compressed table (excluding header), represented
###                     --    using whole 32-bit words, was
###                     --    0% of the size of the code in the body,
###                     --    but only 0% counting the bits really used.
        .global __TT__pip_regTable_end
__TT__pip_regTable_end:
##
## Final Masks (ptr/scal):
## 0004 ........:........:........:........ ........:........:........:XXXXXX..
## 0008 ........:........:........:........ ........:........:..X.....:XXXXXX..
## 000c ........:........:........:........ .X......:........:..X.....:XXXXXX..
## 0010 ........:........:........:........ .X......:........:..X.....:XXXXXX..
## 0014 ........:........:........:........ .X......:...X....:..X.....:XXXXXX..
## 0018 ........:........:........:........ .X......:...X....:..X.....:XXXXXX..
## 001c ........:........:........:........ .X......:...X....:..X.....:XXXXXX..
## 0020 ........:........:........:........ .X......:...X....:..X.....:XXXXXX..
## 0024 ........:........:........:........ .X......:...X..X:..X.....:XXXXXX..
## 0028 ........:........:........:........ .X......:...XX..X:..X.....:XXXXXX..
## 002c ........:........:........:........ .X......:....X..X:..X.....:XXXXXX..
## 0030 ........:........:........:........ .X......:...XX..X:..X.....:XXXXXX..
## 0034 ........:........:........:........ .X......:...XX...:..X.....:XXXXXX..
## 0038 ........:........:........:........ .X......:...XX...:..X.....:XXXXXX..
## 003c ........:........:........:........ .X......:...XX...:..X.....:XXXXXX..
## 0040 ........:........:........:........ .X......:...XXX..:..X.....:XXXXXX..
## 0044 ........:........:........:........ .X......:...XXX.X:..X.....:XXXXXX..
## 0048 ........:........:........:........ .X......:....XX.X:..X.....:XXXXXX..
## 004c ........:........:........:........ .X......:....XX..:..X.....:XXXXXX..
## 0050 ........:........:........:........ .X......:........X:..X.....:XXXXXX..
## 0054 ........:........:........:........ .X......:....X..X:..X.....:XXXXXX..
## 0058 ........:........:........:........ ........:...XX...:..X.....:XXXXXX..
## 005c ........:........:........:........ ........:...XXX..:..X.....:XXXXXX..
## 0060 ........:........:........:........ ........:...XXX..:..X.....:XXXXXX..

[...]

## 0358 ........:........:........:........ .X......:...X....:..X..XXX:XXXXXX..
## 035c ........:........:........:........ ........:...X....:..X..XXX:XXXXXX..
## 0360 ........:........:........:........ .X......:...X....:..X..XXX:XXXXXX..
## 0364 ........:........:........:........ .X......:...XX...:..X..XXX:XXXXXX..
## 0368 ........:........:........:........ ........:...XX..X:..X..XXX:XXXXXX..
## 036c ........:........:........:........ ........:...XXX.X:..X..XXX:XXXXXX..
## 0370 ........:........:........:........ .X......:....XX.X:..X..XXX:XXXXXX..
## 0374 ........:........:........:........ .X......:....XX...:..X..XXX:XXXXXX..
## 0378 ........:........:........:........ ........:....XX...:..X..XXX:XXXXXX..
## 037c ........:........:........:........ .X......:.....X...:..X..XXX:XXXXXX..
## 0380 ........:........:........:........ ........:.....X...:..X..XXX:XXXXXX..
## 0384 ........:........:........:........ .X......:........:..X..XXX:XXXXXX..
## 0388 ........:........:........:........ .X......:...X....:..X..XXX:XXXXXX..
## 038c ........:........:........:........ ........:...X....:..X..XXX:XXXXXX..
## 0390 ........:........:........:........ .X......:...X....:..X..XXX:XXXXXX..
## 0394 ........:........:........:........ .X......:...XX...:..X..XXX:XXXXXX..
## 0398 ........:........:........:........ ........:...XX..X:..X..XXX:XXXXXX..
## 039c ........:........:........:........ .X......:....X..X:..X..XXX:XXXXXX..
## 03a0 ........:........:........:........ .X......:....X...:..X..XXX:XXXXXX..
## 03a4 ........:........:........:........ .X......:....XX...:..X..XXX:XXXXXX..
## 03a8 ........:........:........:........ ........:....XX...:..X..XXX:XXXXXX..
## 03ac ........:........:........:........ .X......:.....X...:..X..XXX:XXXXXX..
```

```
## 03b0 ........:........:........:........ ........:.....X..:..X..XXX:XXXXXX..
## 03b4 ........:........:........:........ .X......:.....X..:..X..XXX:XXXXXX..
## 03b8 ........:........:........:........ ........:.....X..:..X..XXX:XXXXXX..
## 03bc ........:........:........:........ ........:.....X..:..X..XXX:XXXXXX..

[...]

## 0b3c ........:........:........:........ .X......:XXX.XX.X:XXXXXXXX:XXXXXX..
## 0b40 ........:........:........:........ .X......:XXXX.X.X:XXXXXXXX:XXXXXX..
## 0b44 ........:........:........:........ .X......:XXXX.X.X:XXXXXXXX:XXXXXX..
## 0b48 ........:........:........:........ .X......:XXXX.X.X:XXXXXXXX:XXXXXX..
## 0b4c ........:........:........:........ .X......:XXXX.X.X:XXXXXXXX:XXXXXX..
## 0b50 ........:........:........:........ .X......:XXXX.X.X:XXXXXXXX:XXXXXX..
## 0b54 ........:........:........:........ .X......:XXXX.X.X:XXXXXXXX:XXXXXX..
## 0b58 ........:........:........:........ .X......:XXXX.X.X:XXXXXXXX:XXXXXX..
## 0b5c ........:........:........:........ .X......:XXXX.X.X:XXXXXXXX:XXXXXX..
## 0b60 ........:........:........:........ .X......:XXXX.X.X:XXXXXXXX:XXXXXX..
## 0b64 ........:........:........:........ .X......:XXXX.X.X:XXXXXXXX:XXXXXX..
## 0b68 ........:........:........:........ ........:XXXX.X.X:XXXXXXXX:XXXXXX..
## 0b6c ........:........:........:........ ........:XXX..X.X:XXXXXXXX:XXXXXX..
## 0b70 ........:........:........:........ ........:XXX..X.X:XXXXXXX.:XXXXXX..
## 0b74 ........:........:........:........ ........:XXX..X.X:XXXXXX..:XXXXXX..
## 0b78 ........:........:........:........ ........:XXX..X.X:XXXXX...:XXXXXX..
## 0b7c ........:........:........:........ ........:XXX..X.X:XXXX....:XXXXXX..
## 0b80 ........:........:........:........ ........:XXX..X.X:XXX.....:XXXXXX..
## 0b84 ........:........:........:........ ........:XXX..X.X:X.X.....:XXXXXX..
## 0b88 ........:........:........:........ ........:XXX..X.X:..X.....:XXXXXX..
## 0b8c ........:........:........:........ .........:XX..X.X:..X.....:XXXXXX..
## 0b90 ........:........:........:........ .........:..X..X.X:..X.....:XXXXXX..
## 0b94 ........:........:........:........ ........:.....X.X:..X.....:XXXXXX..
## 0b98 ........:........:........:........ ........:.......X:..X.....:XXXXXX..
## 0b9c ........:........:........:........ .X......:........:........:XXXXXX..
## 0ba0 ........:........:........:........ ........:........:........:XXXXXX..
## 0ba4 ........:........:........:........ .X......:........:........:..XXXX..
## 0ba8 ........:........:........:........ ........:........:........:.X.XXX..
## 0bac ........:........:........:........ ........:........:........:..X.XX..
## 0bb0 ........:........:........:........ ........:........:........:..XXXX..
## 0bb4 ........:........:........:........ .X......:........:........:...X.X..
## 0bb8 ........:........:........:........ .X......:........:........:...XXX..
## 0bbc ........:........:........:........ ........:........:........:.X.XX...
## 0bc0 ........:........:........:........ ........:........:........:.X.XXX..
## 0bc4 ........:........:........:........ .X......:........:........:.X.XXX..
## 0bc8 ........:........:........:........ .X......:........:........:..XXX...
## 0bcc ........:........:........:........ .X......:........:........:.X..X...
## 0bd0 ........:........:........:........ .X......:........:........:...X....
## 0bd4 ........:........:........:........ .X......:........:........:...X.X..
## 0bd8 ........:........:........:........ ........:........:........:..X..X..
## 0bdc ........:........:........:........ ........:........:........:..X.XX..
## 0be0 ........:........:........:........ ........:........:........:..XXXX..
## 0be4 ........:........:........:........ .X......:........:........:..XXXX..
## 0be8 ........:........:........:........ .X......:........:........:.X.XX...
## 0bec ........:........:........:........ .X......:........:........:..XX....
## 0bf0 ........:........:........:........ .X......:........:........:....X...
## 0bf4 ........:........:........:........ .X......:........:........:....XX..
## 0bf8 ........:........:........:........ ........:........:........:.X...X..
## 0bfc ........:........:........:........ ........:........:........:.X.X.X..
## 0c00 ........:........:........:........ ........:........:........:.X.XXX..
## 0c04 ........:........:........:........ .X......:........:........:.X.XXX..
## 0c08 ........:........:........:........ .X......:........:........:..XXX...
## 0c0c ........:........:........:........ .X......:........:........:..XXXX..
## 0c10 ........:........:........:........ .X......:........:........:.X..XX..
## 0c14 ........:........:........:........ .X......:........:........:...X.X..
## 0c18 ........:........:........:........ .X......:........:........:...XXX..
## 0c1c ........:........:........:........ ........:........:........:..X.XX..
## 0c20 ........:........:........:........ ........:........:........:..XXXX..
## 0c24 ........:........:........:........ ........:........:........:.X.XX...
## 0c28 ........:........:........:........ .X......:........:........:.X.XX...
## 0c2c ........:........:........:........ .X......:........:........:.X.XXX..
## 0c30 ........:........:........:........ .X......:........:........:..XX.X..
## 0c34 ........:........:........:........ .X......:........:........:....XX..
## 0c38 ........:........:........:........ .X......:........:........:...XXX..
## 0c3c ........:........:........:........ ........:........:........:.X.X.X..
## 0c40 ........:........:........:........ ........:........:........:..XX....
## 0c44 ........:........:........:........ ........:........:........:..XXX...
## 0c48 ........:........:........:........ ........:........:........:.X..X...
## 0c4c ........:........:........:........ .X......:........:........:.X..X...
## 0c50 ........:........:........:........ .X......:........:........:.XX.X...
## 0c54 ........:........:........:........ .X......:........:........:..XX....
## 0c58 ........:........:........:........ ........:........:........:.XX.....
##
```

C.4.3 C using mostly pointers

```
int many_pointer_params(int *a,int *b,int *c,int *d,int *e,int *f,int *g,int *h,int *i,
                int *j,int *k,int *l,int *m,int *n,int *o,int *p,int *q,int *r,
                int *s,int *t,int *u,int *v,int *w,int *x,int *y,int *z)
{
  return *a+*b+*c+*d+*e+*f+*g+*h+*i+*j+*k+*l+*m+*n+*o+*p+*q+*r+*s+*t+*u+*v+*w+*x+*y+*z;
}
```

```
        .global __TT__many_pointer_params_regTable
__TT__many_pointer_params_regTable:
###                     --Save and restore offsets for register window shift
###                       or add/sub offsets for stack pointer adjustment
        .long   0x00000000
        .long   0x00000150
###                     --Start of body and Start of epilogue offsets
        .long   0x00000004
        .long   0x00000148
###                     --Flags: saveRestoreUsed
###                              retIsPtr
###                              spMoved
        .long   0xa0000000
###                     --Frame size:
        .long   0x00000088
###                     --Outgoing params area size:
        .long   0x00000060
###                     --Number of stack slots ever used as ptrs:
        .long   0x00000014
###                     --Used offsets:
        .long   0x000000e4
        .long   0x000000e8
        .long   0x000000ec
        .long   0x000000f0
        .long   0x000000f4
        .long   0x000000f8
        .long   0x000000fc
        .long   0x00000100
        .long   0x00000104
        .long   0x00000108
        .long   0x0000010c
        .long   0x00000110
        .long   0x00000114
        .long   0x00000118
        .long   0x0000011c
        .long   0x00000120
        .long   0x00000124
        .long   0x00000128
        .long   0x0000012c
        .long   0x00000130
###                     --Registers used only as scalars
        .long   0xbfffff03
###                     --Registers used only as pointers
        .long   0x00000000
        .long   0x00000000
###                     --Table columns contain:
### Reg/offs: %g1
### Reg/offs: %i0
### Reg/offs: %i1
### Reg/offs: %i2
### Reg/offs: %i3
### Reg/offs: %i4
### Reg/offs: %i5
### Reg/offs: 228
### Reg/offs: 232
### Reg/offs: 236
### Reg/offs: 240
### Reg/offs: 244
### Reg/offs: 248
### Reg/offs: 252
### Reg/offs: 256
### Reg/offs: 260
### Reg/offs: 264
### Reg/offs: 268
### Reg/offs: 272
### Reg/offs: 276
### Reg/offs: 280
### Reg/offs: 284
### Reg/offs: 288
### Reg/offs: 292
### Reg/offs: 296
### Reg/offs: 300
```

```
### Reg/offs: 304
###                     --Header completed.
###                     --Compressed table:
        .long   0xf055fd07
        .long   0xdc0efc2b
        .long   0x93f0bd01
        .long   0x1fc07784
        .long   0x7fc3edc0
        .long   0x97fc1f78
        .long   0x3bf0c3bf
        .long   0x01dfd07d
        .long   0xc0efc2d7
        .long   0xfc3f60df
        .long   0x90fb70e3
        .long   0xab860018
        .long   0x006006fc
        .long   0x28edc838
        .long   0x0f1ee11c
        .long   0x6d8171e6
        .long   0x15c6985f
        .long   0x19e07c77
        .long   0x86f18e29
        .long   0xc5587718
        .long   0x62dc518b
        .long   0xf13e1fc5
        .long   0xf8cf12e4
        .long   0x1c3d8d71
        .long   0x2645c399
        .long   0x1f0de37c
        .long   0x4792f0c8
###                     -- Table done.
###                     --   861 bits used in compressed table
###                     --    plus two times 52 bits used in the header
###                     --   4212 bits for the full table.
###                     --   compression squeezed down to: 22.9107% of original
###                     --    and 20.4416% excluding the header.
###                     --   Compressed table (excluding header), represented
###                     --    using whole 32-bit words, was
###                     --    33.3333% of the size of the code in the body,
###                     --    but only 33.2176% counting the bits really used.
        .global __TT__many_pointer_params_regTable_end
__TT__many_pointer_params_regTable_end:
##
## Final Masks (ptr/scal):
## 0004 ........:........:........:XXXXXX..:XXXXXXXX:XXXXXXXX:XXXX ........:........:........:........:........:........:....
## 0008 ........:........:........:X.XXXX..:XXXXXXXX:XXXXXXXX:XXXX .X......:........:........:........:........:........:....
## 000c ........:........:........:X..XXX..:XXXXXXXX:XXXXXXXX:XXXX .X......:........:........:.X......:........:........:....
## 0010 ........:........:........:X..XXX..:XXXXXXXX:XXXXXXXX:XXXX ........:........:........:.X......:........:........:....
## 0014 ........:........:........:X..XX...:XXXXXXXX:XXXXXXXX:XXXX .X......:........:........:.X......:........:........:....
## 0018 ........:........:........:X..XXX..:.XXXXXXX:XXXXXXXX:XXXX .X......:........:........:.X......:........:........:....
## 001c ........:........:........:X...XX..:.XXXXXXX:XXXXXXXX:XXXX .X......:........:........:.XX.....:........:........:....
## 0020 ........:........:........:X...XX..:.XXXXXXX:XXXXXXXX:XXXX .X......:........:........:..X.....:........:........:....
## 0024 ........:........:........:X....X..:.XXXXXXX:XXXXXXXX:XXXX .X......:........:........:..XX....:........:........:....
## 0028 ........:........:........:X.......:.XXXXXXX:XXXXXXXX:XXXX .X......:........:.......X:..XX....:........:........:....
## 002c ........:........:........:XX......:..XXXXXX:XXXXXXXX:XXXX .X......:........:.......X:..XX....:........:........:....
## 0030 ........:........:........:XX..X...:..X.XXXX:XXXXXXXX:XXXX .X......:........:.......X:..XX....:........:........:....
## 0034 ........:........:........:XX..XX..:..X.XX.X:XXXXXXXX:XXXX .X......:........:.......X:..XX....:........:........:....
## 0038 ........:........:........:XX..XX..:..X.XX.X:XXXXXXXX:XXXX .X......:........:.......X:...X....:........:........:....
## 003c ........:........:........:XX..XX..:..X.XX.X:XXXXXXXX:XXXX .X......:........:.......X:........:........:........:....
## 0040 ........:........:........:XX..XX..:..X.XX.X:XXXXXXXX:XXXX ........:........:.......X:........:........:........:....
## 0044 ........:........:........:X...XX..:..X.XX.X:XXXXXXXX:XXXX ........:........:......XX:........:........:........:....
## 0048 ........:........:........:X.X.XX..:....XX.X:XXXXXXXX:XXXX ........:........:......XX:........:........:........:....
## 004c ........:........:........:X.X..X..:....XX.X:XXXXXXXX:XXXX ........:........:....X.XX:........:........:........:....
## 0050 ........:........:........:X.XX.X..:.....X.X:XXXXXXXX:XXXX ........:........:....X.XX:........:........:........:....
## 0054 .X......:........:........:X.XX.X..:.......X:XXXXXXXX:XXXX ........:........:....X.XX:........:........:........:....
## 0058 .X......:........:........:X.XX....:.......X:XXXXXXXX:XXXX ........:........:.X..X.XX:........:........:........:....
## 005c .X......:........:........:XXXX....:........:XXXXXXXX:XXXX ........:........:.X..X.XX:........:........:........:....
## 0060 .X......:........:........:XXXXX...:........:X.XXXXXX:XXXX ........:........:.X..X.XX:........:........:........:....
## 0064 .X......:........:........:XXXXXX..:........:X.XX.XXX:XXXX ........:........:.X..X.XX:........:........:........:....
## 0068 .X......:........:........:XX.XXX..:........:X.XX.XXX:XXXX ........:........:.X..XXXX:........:........:........:....
## 006c .X......:........:........:XX..XX..:........:X.XX.XXX:XXXX ........:........:.X.XXXXX:........:........:........:....
## 0070 ........:........:........:XX..XX..:........:X.XX.XXX:XXXX ........:........:.XXXXXXX:........:........:........:....
## 0074 ........:........:........:X...XX..:........:X.XX.XXX:XXXX ........:........:XXXXXXXX:........:........:........:....
## 0078 ........:........:........:X.X.XX..:........:..XX.XXX:XXXX ........:........:XXXXXXXX:........:........:........:....
## 007c ........:........:........:X.X..X..:........:..XX.XXX:XXXX ........:.X......:XXXXXXXX:........:........:........:....
## 0080 ........:........:........:X.XX.X..:........:...X.XXX:XXXX ........:.X......:XXXXXXXX:........:........:........:....
## 0084 .X......:........:........:X.XX.X..:........:.....XXX:XXXX ........:.X......:XXXXXXXX:........:........:........:....
## 0088 .X......:........:........:X.XX....:........:.....XXX:XXXX ........:.X..X...:XXXXXXXX:........:........:........:....
## 008c .X......:........:........:XXXX....:........:......XX:XXXX ........:.X..X...:XXXXXXXX:........:........:........:....
## 0090 .X......:........:........:XXXXX...:........:......X.:XXXX ........:.X..X...:XXXXXXXX:........:........:........:....
## 0094 .X......:........:........:XXXXXX..:........:......X.:XX.X ........:.X..X...:XXXXXXXX:........:........:........:....
```

```
## 0098 .X.....:.......:........:XX.XXX..:........:......X.:XX.X ........:XX..X...:XXXXXXXX:........:.......:........:....
## 009c .X.....:.......:........:XX..XX..:........:......X.:XX.X ........:XXX.X...:XXXXXXXX:........:.......:........:....
## 00a0 ........:.......:........:XX..XX..:........:......X.:XX.X ........:XXXXX...:XXXXXXXX:........:.......:........:....
## 00a4 ........:.......:........:X...XX..:........:......X.:XX.X ........:XXXXXX..:XXXXXXXX:........:.......:........:....
## 00a8 ........:.......:........:X.X.XX..:........:........:XX.X ........:XXXXXX..:XXXXXXXX:........:.......:........:....
## 00ac ........:.......:........:X.X..X..:........:........:XX.X ........:XXXXXX..:XXXXXXXX:.X......:.......:........:....
## 00b0 ........:.......:........:X.XX.X..:........:........:.X.X ........:XXXXXX..:XXXXXXXX:.X......:.......:........:....
## 00b4 .X......:.......:........:X.XX.X..:........:........:...X ........:XXXXXX..:XXXXXXXX:.X......:.......:........:....
## 00b8 .X......:.......:........:X.XX....:........:........:...X ........:XXXXXX..:XXXXXXXX:.X..X...:.......:........:....
## 00bc .X......:.......:........:X.XX.X..:........:........:.... ........:XXXXXX..:XXXXXXXX:.X..X...:.......:........:....
## 00c0 .X......:.......:........:X..X.X..:........:........:.... ........:XXXXXX.X:XXXXXXXX:.X..X...:.......:........:....
## 00c4 .X......:.......:........:...X.X..:........:........:.... ........:XXXXXX.X:XXXXXXXX:XX..X...:.......:........:....
## 00c8 .X......:.......:........:....X...:........:........:.... ........:XXXXXX.X:XXXXXXXX:XXX.X...:.......:........:....
## 00cc ........:.......:........:....X...:........:........:.... ........:XXXXXX.X:XXXXXXXX:XXXXX...:.......:........:....
## 00d0 ........:.......:........:.......:........:........:.... .X......:XXXXXX.X:XXXXXXXX:XXXXX...:.......:........:....
## 00d4 ........:.......:........:........:........:........:.... .X......:XXXXXX.X:XXXXXXXX:XXXXXX..:.......:........:....
## 00d8 ........:.......:........:........:........:........:.... .X......:XXXXXX.X:XXXXXXXX:XXXXX...:.......:........:....
## 00dc ........:.......:........:.......:........:........:.... .X......:XXXXXX.X:XXXXXXXX:XXXXXX..:.......:........:....
## 00e0 ........:.......:........:........:........:........:.... .X......:XXXXXX.X:XXXXXXXX:XXXXX...:.......:........:....
## 00e4 ........:.......:........:........:........:........:.... .X......:XXXXXX.X:XXXXXXXX:XXXXXX..:.......:........:....
## 00e8 ........:.......:........:........:........:........:.... .X......:XXXXXX.X:XXXXXXXX:XXXXX...:.......:........:....
## 00ec ........:.......:........:.......:........:........:.... .X......:XXXXXX.X:XXXXXXXX:XXXXXX..:.......:........:....
## 00f0 ........:.......:........:........:........:........:.... .X......:XXXXXX.X:XXXXXXXX:XXXXX...:.......:........:....
## 00f4 ........:.......:........:........:........:........:.... .X......:XXXXXX.X:XXXXXXXX:XXXXXX..:.......:........:....
## 00f8 ........:.......:........:........:........:........:.... .X......:XXXXXX.X:XXXXXXXX:XXXXX...:.......:........:....
## 00fc ........:.......:........:.......:........:........:.... .X......:XXXXXX.X:XXXXXXX.:XXXXX...:.......:........:....
## 0100 ........:.......:........:........:........:........:.... .X......:XXXXXX.X:XXXXXX..:.XXXXX..:.......:........:....
## 0104 ........:.......:........:........:........:........:.... .X......:XXXXXX.X:XXXXX...:XXXXX...:.......:........:....
## 0108 ........:.......:........:.......:........:........:.... .X......:XXXXXX.X:XXXX....:.XXXXX..:.......:........:....
## 010c ........:.......:........:........:........:........:.... .X......:XXXXXX.X:XXX.....:XXXXX...:.......:........:....
## 0110 ........:.......:........:........:........:........:.... .X......:XXXXXX.X:XX......:.XXXXX..:.......:........:....
## 0114 ........:.......:........:........:........:........:.... .X......:XXXXXX.X:X.......:XXXXX...:.......:........:....
## 0118 ........:.......:........:.......:........:........:.... .X......:XXXXXX.X:........:.XXXXX..:.......:........:....
## 011c ........:.......:........:........:........:........:.... .X......:.XXXXX.X:........:XXXXX...:.......:........:....
## 0120 ........:.......:........:........:........:........:.... .X......:..XXXX.X:........:.XXXXX..:.......:........:....
## 0124 ........:.......:........:........:........:........:.... .X......:...XXX.X:........:XXXXX...:.......:........:....
## 0128 ........:.......:........:.......:........:........:.... .X......:....XX.X:........:.XXXXX..:.......:........:....
## 012c ........:.......:........:........:........:........:.... .X......:.....X.X:........:XXXXX...:.......:........:....
## 0130 ........:.......:........:........:........:........:.... .X......:......X:........:.XXXXX..:.......:........:....
## 0134 ........:.......:........:........:........:........:.... .X......:.......:........:XXXXX...:.......:........:....
## 0138 ........:.......:........:.......:........:........:.... .X......:.......:........:..XXXX..:.......:........:....
## 013c ........:.......:........:........:........:........:.... .X......:.......:........:.X.XX...:.......:........:....
## 0140 ........:.......:........:........:........:........:.... .X......:.......:........:..X.X...:.......:........:....
## 0144 ........:.......:........:........:........:........:.... .X......:.......:........:.X......:.......:........:....
##
```

C.4.4 Ada

```
procedure add(tree:in Pack1_Ptr;n:in Pack1_Ptr) is

target: Pack1_Ptr;
workLeft: Boolean;
x:Int;

begin
  workLeft:=true;
  x:=n.all.data;
  target:=tree;

  while workLeft loop
   if target.all.data > x then
    if target.all.l /= null then
     target:=target.all.l;
    else
     target.all.l:=n;
     workLeft:=false;
    end if;
   else
    if target.all.r /= null then
     target:=target.all.r;
    else
     target.all.r:=n;
     workLeft:=false;
    end if;
   end if;
  end loop;
end add;
```

```
        .global __TT__integral__add_regTable
__TT__integral__add_regTable:
###                     --Save and restore offsets for register window shift
###                       or add/sub offsets for stack pointer adjustment
        .long   0x00000000
        .long   0x00000000
###                     --Start of body and Start of epilogue offsets
        .long   0x00000000
        .long   0x00000054
###                     --Flags: saveRestoreUsed
###                              retIsPtr
###                              spMoved
        .long   0x00000000
###                     --Frame size:
        .long   0x00000000
###                     --Outgoing params area size:
        .long   0x00000000
###                     --Number of stack slots ever used as ptrs:
        .long   0x00000000
###                     --Used offsets:
###                     --Registers used only as scalars
        .long   0xbf3fffff
###                     --Registers used only as pointers
        .long   0x00400000
###                     --Table columns contain:
### Reg/offs: %g1
### Reg/offs: %o0
###                     --Header completed.
###                     --Compressed table:
        .long   0x00e383fe
        .long   0xfbe00000
###                     -- Table done.
###                     --   43 bits used in compressed table
###                     --    plus two times 32 bits used in the header
###                     --   672 bits for the full table.
###                     --   compression squeezed down to: 15.9226% of original
###                     --    and 6.39881% excluding the header.
###                     --   Compressed table (excluding header), represented
###                     --    using whole 32-bit words, was
###                     --    9.52381% of the size of the code in the body,
###                     --    but only 6.39881% counting the bits really used.
        .global __TT__integral__add_regTable_end
__TT__integral__add_regTable_end:
##
## Final Masks (ptr/scal):
## 0000 ........:XX......:........:........ ........:........:........:........
## 0004 ........:XX......:........:........ ........:....X...:........:........
## 0008 ........:XX......:........:........ ........:....XX..:........:........
## 000c ........:XX......:........:........ .X......:....XX..:........:........
## 0010 ........:XX......:........:........ ........:....XX..:........:........
## 0014 ........:XX......:........:........ ........:....XX..:........:........
## 0018 ........:XX......:........:........ ........:....XX..:........:........
## 001c .X......:XX......:........:........ ........:....XX..:........:........
## 0020 .X......:XX......:........:........ ........:....XX..:........:........
## 0024 .X......:.X......:........:........ ........:....XX..:........:........
## 0028 ........:XX......:........:........ ........:....X...:........:........
## 002c ........:XX......:........:........ ........:....X...:........:........
## 0030 ........:XX......:........:........ ........:....X...:........:........
## 0034 .X......:XX......:........:........ ........:....XX..:........:........
## 0038 .X......:XX......:........:........ ........:....XX..:........:........
## 003c .X......:.X......:........:........ ........:....XX..:........:........
## 0040 ........:XX......:........:........ ........:....X...:........:........
## 0044 ........:XX......:........:........ ........:....X...:........:........
## 0048 ........:XX......:........:........ ........:....XX..:........:........
## 004c ........:XX......:........:........ ........:....XX..:........:........
## 0050 ........:XX......:........:........ ........:....XX..:........:........
##
```

C.4.5 Pascal

```
procedure pascalAdd(root,n:nodeptr);
var
  target: nodeptr;
  workLeft: boolean;
  x: integer;

begin
  workLeft:=true;
```

```
  x:=n^.data;
  target:=root;

  printInt(x);
  while workLeft do begin
   if target^.data > x then begin
    if not (target^.l = nil) then
     target:=target^.l
    else begin
     target^.l:=n;
     workLeft:=false
    end
   end else begin
    if not (target^.r = nil) then
     target:=target^.r
    else begin
     target^.r:=n;
     workLeft:=false
    end
   end
  end
end;
```

```
        .global __TT__Pascaladd_regTable
__TT__Pascaladd_regTable:
###                     --Save and restore offsets for register window shift
###                       or add/sub offsets for stack pointer adjustment
        .long   0x00000000
        .long   0x00000068
###                     --Start of body and Start of epilogue offsets
        .long   0x00000004
        .long   0x00000060
###                     --Flags: saveRestoreUsed
###                              retIsPtr
###                              spMoved
        .long   0xa0000000
###                     --Frame size:
        .long   0x00000070
###                     --Outgoing params area size:
        .long   0x00000060
###                     --Number of stack slots ever used as ptrs:
        .long   0x00000000
###                     --Used offsets:
###                     --Registers used only as scalars
        .long   0xbfffff3f
###                     --Registers used only as pointers
        .long   0x00000040
###                     --Table columns contain:
### Reg/offs: %g1
### Reg/offs: %i0
###                     --Header completed.
###                     --Compressed table:
        .long   0x00071cff
        .long   0xfdf60000
###                     -- Table done.
###                     --   47 bits used in compressed table
###                     --    plus two times 32 bits used in the header
###                     --   736 bits for the full table.
###                     --   compression squeezed down to: 15.0815% of original
###                     --    and 6.38587% excluding the header.
###                     --   Compressed table (excluding header), represented
###                     --    using whole 32-bit words, was
###                     --    8.69565% of the size of the code in the body,
###                     --    but only 6.38587% counting the bits really used.
        .global __TT__Pascaladd_regTable_end
__TT__Pascaladd_regTable_end:
##
## Final Masks (ptr/scal):
## 0004 ........:........:........:XX..... ........:........:........:........
## 0008 ........:........:........:XX..... ........:........:X.......:........
## 000c ........:........:........:XX..... ........:........:X.......:........
## 0010 ........:........:........:XX..... ........:........:X.......:........
## 0014 ........:........:........:XX..... ........:.....X..:X.......:........
## 0018 ........:........:........:XX..... ........:.....X..:X.......:........
## 001c ........:........:........:XX..... ........:.....X..:X.......:........
## 0020 ........:........:........:XX..... ........:.....X..:X.......:........
## 0024 ........:........:........:XX..... ........:X....X..:X.......:........
## 0028 ........:........:........:XX..... ........:.....X..:X.......:........
## 002c ........:........:........:XX..... ........:.....X..:X.......:........
## 0030 ........:........:........:XX..... ........:.....X..:X.......:........
## 0034 .X......:........:........:XX..... ........:.....X..:X.......:........
```

```
## 0038 .X......:........:........:XX..... ........:.....X..:X.......:........
## 003c .X......:........:........:.X..... ........:.....X..:X.......:........
## 0040 ........:........:........:XX..... ........:........:X.......:........
## 0044 ........:........:........:XX..... ........:........:X.......:........
## 0048 ........:........:........:XX..... ........:........:X.......:........
## 004c .X......:........:........:XX..... ........:.....X..:X.......:........
## 0050 .X......:........:........:XX..... ........:.....X..:X.......:........
## 0054 .X......:........:........:.X..... ........:.....X..:X.......:........
## 0058 ........:........:........:XX..... ........:........:X.......:........
## 005c ........:........:........:XX..... ........:........:X.......:........
##
```

C.4.6 C using various expressions

```
void rather_complex_test(int a,int *b)
{
  char *c=some_char(),*d=some_char();
  int o1=a,o2=a-3,o3=a+o1;
  int *po1=some_int();
  type_a *ta1=some_a();
  type_b *ta2=some_b();

  o1=1;
  do
  {
   int i,j=0;

   for (i=ping(b);i<ping(poing(j));i++) {
    peng(a+ping(pang(a,b)));
    while (pong(poing(a),pong(b,1))) {
     if (j<91) {
      ta2->cc=1+(*(ta1->cp=some_char()));
      peng(a-1);
      o3=-19;
      j=pong(b,a+2) ? a*ping(b) : 0;
     } else {
      j=ping(pang(j,b));
      break;
     }
    }
   }
   fun2(o1,o2,o3,o2,o1,ping(poing(o2)),143);
   while (pong(poing(a),pong(b,1))) {
    if (j<91) {
     peng(a-ta2->u2);
    } else {
     j=ping(pang(j,b));
     break;
    }
    ta1=fun1(o2-8,ta2,*c,*po1,b);
   }
  } while (some_b()!=fun3(some_int(),*po1+ping(pang(3,ta1->p)),
                          po1,ta1,some_b(),d));
}

        .global __TT__rather_complex_test_regTable
__TT__rather_complex_test_regTable:
###                     --Save and restore offsets for register window shift
###                       or add/sub offsets for stack pointer adjustment
        .long   0x00000000
        .long   0x0000022c
###                     --Start of body and Start of epilogue offsets
        .long   0x00000004
        .long   0x00000224
###                     --Flags: saveRestoreUsed
###                              retIsPtr
###                              spMoved
        .long   0xa0000000
###                     --Frame size:
        .long   0x00000078
###                     --Outgoing params area size:
        .long   0x00000068
###                     --Number of stack slots ever used as ptrs:
        .long   0x00000000
###                     --Used offsets:
###                     --Registers used only as scalars
        .long   0xff0706b3
###                     --Registers used only as pointers
```

```
        .long   0x00000040
###                     --Table columns contain:
### Reg/offs: %o0
### Reg/offs: %o1
### Reg/offs: %o2
### Reg/offs: %o3
### Reg/offs: %o4
### Reg/offs: %l0
### Reg/offs: %l1
### Reg/offs: %l2
### Reg/offs: %l3
### Reg/offs: %l4
### Reg/offs: %l7
### Reg/offs: %i4
### Reg/offs: %i5
###                     --Header completed.
###                     --Compressed table:
        .long   0xc4f7b460
        .long   0x607c04dc
        .long   0x10c07c02
        .long   0xdc11665e
        .long   0x70cbf07b
        .long   0x70e370e5
        .long   0x9903f433
        .long   0xa73d373c
        .long   0x3b70cbb7
        .long   0x67605ca8
        .long   0xedc1b847
        .long   0x026e7c03
        .long   0xa7700e24
        .long   0x001bb403
        .long   0xbb86f800
###                     -- Table done.
###                     --   471 bits used in compressed table
###                     --    plus two times 32 bits used in the header
###                     --   4352 bits for the full table.
###                     --   compression squeezed down to: 12.2932% of original
###                     --    and 10.8226% excluding the header.
###                     --   Compressed table (excluding header), represented
###                     --    using whole 32-bit words, was
###                     --    11.0294% of the size of the code in the body,
###                     --    but only 10.8226% counting the bits really used.
        .global __TT__rather_complex_test_regTable_end
__TT__rather_complex_test_regTable_end:
##
## Final Masks (ptr/scal):
## 0004 ........:.......:........:.X..... ........:.......:........:X.......
## 0008 ........:.......:........:.X..... ........:.......:........:X.......
## 000c ........:X......:........:.X..... ........:.......:......X.:X.......
## 0010 ........:.......:........:.X...X.. ........:.......:......X.:X.......
## 0014 ........:.......:........:.X...X.. ........:.......:......X.:X.......
## 0018 ........:X......:........:.X...X.. ........:.......:.....XX.:X.......
## 001c ........:X......:........:.X...X.. ........:.......:.....XX.:X.......
## 0020 ........:X......:........:.X..XX.. ........:.......:.....XX.:X.......
## 0024 ........:X......:........:.X..XX.. ........:.......:.....XX.:X.......
## 0028 ........:X......:.......X:.X..XX.. ........:.......:.....XX.:X.......
## 002c ........:X......:.......X:.X..XX.. ........:.......:.....XX.:X.......
## 0030 ........:X......:....X..X:.X..XX.. ........:.......:.....XX.:X.......
## 0034 ........:.......:...XX..X:.X..XX.. ........:.......:.....XX.:X.......
## 0038 ........:X......:...XX..X:.X..XX.. ........:.......:.....XX.:X.......
## 003c ........:X......:...XX..X:.X..XX.. ........:.......:.....XX.:X.......
## 0040 ........:.......:...XX..X:.X..XX.. ........:X......:.X...XX.:X.......
## 0044 ........:.......:...XX..X:.X..XX.. ........:.......:.XX..XX.:X.......
## 0048 ........:.......:...XX..X:.X..XX.. ........:.......:.XX..XX.:X.......
## 004c ........:X......:...XX..X:.X..XX.. ........:.......:.XX..XX.:X.......
## 0050 ........:X......:...XX..X:.X..XX.. ........:.......:.XX..XX.:X.......
## 0054 ........:.......:...XX..X:.X..XX.. ........:X......:.XX..XX.:X.......
## 0058 ........:.......:...XX..X:.X..XX.. ........:.......:.XX..XX.:X.......
## 005c ........:.......:...XX..X:.X..XX.. ........:.......:.XX..XX.:X.......
## 0060 ........:.X.....:...XX..X:.X..XX.. ........:.......:.XX..XX.:X.......
## 0064 ........:.X.....:...XX..X:.X..XX.. ........:.......:.XX..XX.:X.......
## 0068 ........:X......:...XX..X:.X..XX.. ........:.......:.XX..XX.:X.......
## 006c ........:X......:...XX..X:.X..XX.. ........:.......:.XX..XX.:X.......
## 0070 ........:.......:...XX..X:.X..XX.. ........:X......:.XX..XX.:X.......
## 0074 ........:.......:...XX..X:.X..XX.. ........:X......:.XX..XX.:X.......
## 0078 ........:.......:...XX..X:.X..XX.. ........:.......:.XX..XX.:X.......
## 007c ........:.......:...XX..X:.X..XX.. ........:.......:.XX..XX.:X.......
## 0080 ........:X......:...XX..X:.X..XX.. ........:.......:.XX..XX.:X.......
## 0084 ........:X......:...XX..X:.X..XX.. ........:.X.....:.XX..XX.:X.......
## 0088 ........:.......:X..XX..X:.X..XX.. ........:.X.....:.XX..XX.:X.......
## 008c ........:.......:X..XX..X:.X..XX.. ........:.X.....:.XX..XX.:X.......
```

```
## 0090 ........:........:X..XX..X:.X..XX.. ........:X......:.XX..XX.:X.......
## 0094 ........:........:X..XX..X:.X..XX.. ........:.X.....:.XX..XX.:X.......
## 0098 ........:........:X..XX..X:.X..XX.. ........:.X.....:.XX..XX.:X.......
## 009c ........:........:...XX..X:.X..XX.. ........:X......:.XX..XX.:X.......
## 00a0 ........:........:...XX..X:.X..XX.. ........:........:.XX..XX.:X.......
## 00a4 ........:........:...XX..X:.X..XX.. ........:........:.XX..XX.:X.......
## 00a8 ........:........:...XX..X:.X..XX.. ........:........:.XX..XX.:X.......
## 00ac ........:........:...XX..X:.X..XX.. ........:........:.XX..XX.:X.......
## 00b0 ........:........:...XX..X:.X..XX.. ........:........:..X...X.:X.......
## 00b4 ........:........:...XX..X:.X..XX.. ........:........:..X...X.:X.......
## 00b8 ........:X.......:...XX..X:.X..XX.. ........:........:..X..XX.:X.......
## 00bc ........:X.......:...XX..X:.X..XX.. ........:........:..X..XX.:X.......
## 00c0 ........:........:...XX..X:.X..XX.. .........:..X.....:..X..XX.:X.......
## 00c4 ........:........:...XX..X:.X..XX.. ........:X.X.....:..X..XX.:X.......
## 00c8 ........:........:...XX..X:.X..XX.. ........:XX......:..X..XX.:X.......
## 00cc ........:........:...XX..X:.X..XX.. ........:XX......:..X..XX.:X.......
## 00d0 ........:........:...XX..X:.X..XX.. ........:........:..X..XX.:X.......
## 00d4 ........:........:...XX..X:.X..XX.. .........:.X.....:..X..XX.:X.......
## 00d8 ........:........:...XX..X:.X..XX.. .........:.X.....:..X..XX.:X.......
## 00dc ........:........:...XX..X:.X..XX.. .........:X......:..X..XX.:X.......
## 00e0 ........:........:...XX..X:.X..XX.. ........:........:..X..XX.:X.......
## 00e4 ........:........:...XX..X:.X..XX.. ........:........:..X..XX.:X.......
## 00e8 ........:........:...XX..X:.X..XX.. ........:........:..X..XX.:X.......
## 00ec ........:........:...XX..X:.X..XX.. ........:........:..X..XX.:X.......
## 00f0 ........:........:...XX..X:.X..XX.. .........:X......:..X..XX.:X.......
## 00f4 ........:........:...XX..X:.X..XX.. .........:.X.....:..X..XX.:X.......
## 00f8 ........:........:...XX..X:.X..XX.. .........:.X.....:..X..XX.:X.......
## 00fc ........:........:...XX..X:.X..XX.. .........:X......:..X..XX.:X.......
## 0100 ........:........:...XX..X:.X..XX.. .........:X......:..X..XX.:X.......
## 0104 .........:X......:...XX..X:.X..XX.. ........:........:.XX..XX.:X.......
## 0108 .........:.X.....:...XX..X:.X..XX.. ........:........:.XX..XX.:X.......
## 010c .........:X......:...XX..X:.X..XX.. ........:........:..X..XX.:X.......
## 0110 .........:X......:...XX..X:.X..XX.. ........:........:..X..XX.:X.......
## 0114 ........:........:...XX..X:.X..XX.. .........:X......:..X..XX.:X.......
## 0118 ........:........:...XX..X:.X..XX.. ........:........:.XX..XX.:X.......
## 011c ........:........:...XX..X:.X..XX.. ........:........:.XX..XX.:X.......
## 0120 ........:........:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 0124 ........:........:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 0128 .........:X......:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 012c .........:X......:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 0130 ........:........:...XX..X:.X..XX.. .........:X......:.X...XX.:X.......
## 0134 ........:........:...XX..X:.X..XX.. ........:.....X..:.X...XX.:X.......
## 0138 ........:........:...XX..X:.X..XX.. .........:X....X..:.X...XX.:X.......
## 013c ........:........:...XX..X:.X..XX.. .........:XX...X..:.X...XX.:X.......
## 0140 ........:........:...XX..X:.X..XX.. .........:.X...X..:.X...XX.:X.......
## 0144 ........:........:...XX..X:.X..XX.. .........:.XX..X..:.X...XX.:X.......
## 0148 ........:........:...XX..X:.X..XX.. .........:XXX..X..:.X...XX.:X.......
## 014c ........:........:...XX..X:.X..XX.. .........:XXXX.X..:.X...XX.:X.......
## 0150 ........:........:...XX..X:.X..XX.. .........:XXXX.X..:.X...XX.:X.......
## 0154 ........:........:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 0158 ........:........:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 015c .........:X......:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 0160 .........:X......:...XX..X:.X..XX.. .........:.X.....:.X...XX.:X.......
## 0164 ........:........:..XXX..X:.X..XX.. .........:.X.....:.X...XX.:X.......
## 0168 ........:........:..XXX..X:.X..XX.. .........:.X.....:.X...XX.:X.......
## 016c ........:........:..XXX..X:.X..XX.. .........:X......:.X...XX.:X.......
## 0170 ........:........:..XXX..X:.X..XX.. .........:.X.....:.X...XX.:X.......
## 0174 ........:........:..XXX..X:.X..XX.. .........:.X.....:.X...XX.:X.......
## 0178 ........:........:...XX..X:.X..XX.. .........:X......:.X...XX.:X.......
## 017c ........:........:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 0180 ........:........:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 0184 ........:........:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 0188 ........:........:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 018c ........:........:...X...X:.X..XX.. ........:........:.X...XX.:X.......
## 0190 ........:........:...X...X:.X..XX.. .X......:........:.X...XX.:X.......
## 0194 ........:........:...X...X:.X..XX.. .X......:........:.X...XX.:X.......
## 0198 ........:........:...X...X:.X..XX.. ........:........:.X...XX.:X.......
## 019c ........:........:...X...X:.X..XX.. .........:..X.....:.X...XX.:X.......
## 01a0 ........:........:...X...X:.X..XX.. .........:..XX....:.X...XX.:X.......
## 01a4 ........:........:...X...X:.X..XX.. .........:X.XX....:.X...XX.:X.......
## 01a8 .........:.X.....:...X...X:.X..XX.. .........:X.XX....:.X...XX.:X.......
## 01ac .........:.X.....:...X...X:.X..XX.. .........:X.XX....:.X...XX.:X.......
## 01b0 .........:X......:...X...X:.X..XX.. ........:........:.X...XX.:X.......
## 01b4 .........:X......:...X...X:.X..XX.. ........:........:.X...XX.:X.......
## 01b8 .........:.X.....:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 01bc .........:.X.....:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 01c0 .........:X......:...XX..X:.X..XX.. ........:........:.....XX.:X.......
## 01c4 .........:X......:...XX..X:.X..XX.. ........:........:.....XX.:X.......
## 01c8 ........:........:...XX..X:.X..XX.. ........:........:.....XX.:X.......
## 0144 ........:........:...XX..X:.X..XX.. .........:.XX..X..:.X...XX.:X.......
## 0148 ........:........:...XX..X:.X..XX.. .........:XXX..X..:.X...XX.:X.......
```

```
## 014c ........:........:...XX..X:.X..XX.. ........:XXXX.X..:.X...XX.:X.......
## 0150 ........:........:...XX..X:.X..XX.. ........:XXXX.X..:.X...XX.:X.......
## 0154 ........:........:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 0158 ........:........:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 015c ........:X.......:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 0160 ........:X.......:...XX..X:.X..XX.. ........:.X......:.X...XX.:X.......
## 0164 ........:........:..XXX..X:.X..XX.. ........:.X......:.X...XX.:X.......
## 0168 ........:........:..XXX..X:.X..XX.. ........:.X......:.X...XX.:X.......
## 016c ........:........:..XXX..X:.X..XX.. ........:X.......:.X...XX.:X.......
## 0170 ........:........:..XXX..X:.X..XX.. ........:.X......:.X...XX.:X.......
## 0174 ........:........:..XXX..X:.X..XX.. ........:.X......:.X...XX.:X.......
## 0178 ........:........:...XX..X:.X..XX.. ........:X.......:.X...XX.:X.......
## 017c ........:........:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 0180 ........:........:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 0184 ........:........:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 0188 ........:........:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 018c ........:........:...X...X:.X..XX.. ........:........:.X...XX.:X.......
## 0190 ........:........:...X...X:.X..XX.. .X......:........:.X...XX.:X.......
## 0194 ........:........:...X...X:.X..XX.. .X......:........:.X...XX.:X.......
## 0198 ........:........:...X...X:.X..XX.. ........:........:.X...XX.:X.......
## 019c ........:........:...X...X:.X..XX.. ........:..X.....:.X...XX.:X.......
## 01a0 ........:........:...X...X:.X..XX.. ........:..XX....:.X...XX.:X.......
## 01a4 ........:........:...X...X:.X..XX.. ........:X.XX....:.X...XX.:X.......
## 01a8 ........:.X......:...X...X:.X..XX.. ........:X.XX....:.X...XX.:X.......
## 01ac ........:.X......:...X...X:.X..XX.. ........:X.XX....:.X...XX.:X.......
## 01b0 ........:X.......:...X...X:.X..XX.. ........:........:.X...XX.:X.......
## 01b4 ........:X.......:...X...X:.X..XX.. ........:........:.X...XX.:X.......
## 01b8 ........:.X......:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 01bc ........:.X......:...XX..X:.X..XX.. ........:........:.X...XX.:X.......
## 01c0 ........:X.......:...XX..X:.X..XX.. ........:........:....XX.:X.......
## 01c4 ........:X.......:...XX..X:.X..XX.. ........:........:....XX.:X.......
## 01c8 ........:........:...XX..X:.X..XX.. ........:........:....XX.:X.......
## 01cc ........:........:...XX..X:.X..XX.. ........:........:....XX.:X.......
## 01d0 ........:X.......:...XX..X:.X..XX.. ........:........:....XX.:X.......
## 01d4 ........:X.......:...XX..X:.X..XX.. ........:........:....XX.:X.......
## 01d8 ........:X.......:..XXX..X:.X..XX.. ........:........:....XX.:X.......
## 01dc ........:XX......:..XXX..X:.X..XX.. ........:........:....XX.:X.......
## 01e0 ........:.X......:.XXXX..X:.X..XX.. ........:........:....XX.:X.......
## 01e4 ........:.X......:.XXXX..X:.X..XX.. ........:........:....XX.:X.......
## 01e8 ........:X.......:.XXXX..X:.X..XX.. ........:........:....XX.:X.......
## 01ec ........:X.......:.XXXX..X:.X..XX.. ........:........:....XX.:X.......
## 01f0 ........:........:.XXXX..X:.X..XX.. ........:X.......:....XX.:X.......
## 01f4 ........:........:.XXXX..X:.X..XX.. ........:XX......:....XX.:X.......
## 01f8 ........:........:.XXXX..X:.X..XX.. ........:XX......:....XX.:X.......
## 01fc ........:X.......:.XXXX..X:.X..XX.. ........:........:X...XX.:X.......
## 0200 ........:X.......:.XXXX..X:.X..XX.. ........:.X......:....XX.:X.......
## 0204 ........:....X...:.XXXX..X:.X..XX.. ........:.X......:....XX.:X.......
## 0208 ........:..X.X...:.XXXX..X:.X..XX.. ........:.X......:....XX.:X.......
## 020c ........:X.X.X...:..XXX..X:.X..XX.. ........:.X......:....XX.:X.......
## 0210 ........:X.XXX...:..XXX..X:.X..XX.. ........:.X......:....XX.:X.......
## 0214 ........:X.XXX...:..XXX..X:.X..XX.. ........:.X......:....XX.:X.......
## 0218 ........:X.......:..XXX..X:.X..XX.. ........:........:....XX.:X.......
## 021c ........:........:...XX..X:.X..XX.. ........:........:....XX.:X.......
## 0220 ........:........:...XX..X:.X..XX.. ........:........:....XX.:X.......
##
```

C.4.7 C++

```
class hello {
public:
  int a;
  int b;
  hello *h;
  virtual void something();
  void somethingElse(hello*);
};

main()
{
  hello h1,h2;
  h1.something();
  h2.somethingElse(&h1);
  somethingExternal(&h2);
}
         .global __TT__main_regTable
__TT__main_regTable:
###                     --Save and restore offsets for register window shift
###                       or add/sub offsets for stack pointer adjustment
```

```
        .long   0x00000000
        .long   0x00000054
###                     --Start of body and Start of epilogue offsets
        .long   0x00000004
        .long   0x0000004c
###                     --Flags: saveRestoreUsed
###                              retIsPtr
###                              spMoved
        .long   0xa0000000
###                     --Frame size:
        .long   0x00000090
###                     --Outgoing params area size:
        .long   0x00000060
###                     --Number of stack slots ever used as ptrs:
        .long   0x00000000
###                     --Used offsets:
###                     --Registers used only as scalars
        .long   0xff67ffff
###                     --Registers used only as pointers
        .long   0x00000000
        .long   0x00000000
###                     --Table columns contain:
### Reg/offs: %o0
### Reg/offs: %o3
### Reg/offs: %o4
###                     --Header completed.
###                     --Compressed table:
        .long   0x03e3e300
        .long   0x01ff8000
###                     -- Table done.
###                     --   55 bits used in compressed table
###                     --    plus two times 36 bits used in the header
###                     --   648 bits for the full table.
###                     --   compression squeezed down to: 19.5988% of original
###                     --    and 8.48765% excluding the header.
###                     --   Compressed table (excluding header), represented
###                     --    using whole 32-bit words, was
###                     --    11.1111% of the size of the code in the body,
###                     --    but only 9.54861% counting the bits really used.
        .global __TT__main_regTable_end
__TT__main_regTable_end:
##
## Final Masks (ptr/scal):
## 0004 ........:........:........:........ ........:........:........:........
## 0008 ........:........:........:........ .X......:........:........:........
## 000c ........:....X...:........:........ .X......:........:........:........
## 0010 ........:...XX...:........:........ .X......:........:........:........
## 0014 ........:...XX...:........:........ .X......:..X.....:........:........
## 0018 ........:X...X...:........:........ .X......:..X.....:........:........
## 001c ........:X...X...:........:........ ........:..X.....:........:........
## 0020 ........:X...X...:........:........ ........:.XX.....:........:........
## 0024 ........:X...X...:........:........ .X......:.XX.....:........:........
## 0028 ........:X...X...:........:........ .X......:.XX.....:........:........
## 002c ........:....X...:........:........ .X......:.XX.....:........:........
## 0030 ........:....X...:........:........ .X......:.XX.....:........:........
## 0034 ........:........:........:........ .X......:.XX.....:........:........
## 0038 ........:X.......:........:........ .X......:.XX.....:........:........
## 0034 ........:........:........:........ .X......:.XX.....:........:........
## 0038 ........:X.......:........:........ .X......:.XX.....:........:........
## 003c ........:X.......:........:........ .X......:.X......:........:........
## 0040 ........:X.......:........:........ .X......:........:........:........
## 0044 ........:X.......:........:........ ........:........:........:........
## 0048 ........:X.......:........:........ ........:........:........:........
##
```

"Etaoin Shrdlu"

Bibliography

[AAB+00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000, http://www.research.ibm.com/journal/sj/391/alpern.html.

[AB01] G. Antoniu and L. Boug. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. In *Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01)*, San Francisco, April 2001. http://www.inria.fr/rrrt/rr-4108.html.

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, 1983, http://www.dcs.st-and.ac.uk/research/publications/ABC+83a.php.

[ABN99] Gabriel Antoniu, Luc Bouge, and Raymond Namyst. An efficient and transparent thread migration scheme in the PM2 runtime system. Technical Report RR-3610, Inria, Institut National de Recherche en Informatique et en Automatique, January 1999, http://www.inria.fr/rrrt/rr-3610.html.

[ABNP01] Gabriel Antoniu, Luc Bougé, Raymond Namyst, and Christian Pérez. Compiling data-parallel programs to A distributed runtime environment with thread isomigration. *Parallel Processing Letters*, 10(2-3):201–214, June 2001, http://perso.ens-lyon.fr/alain.darte/cpc2000/antoniu.html. Special issue on Compilers for Parallel Computers (CPC 2000).

[ACL+99] Bowen Alpern, Anthony Cocchi, Derek Lieber, Mark Mergen, and Vivek Sarkar. Jalapeño — a compiler-supported Java virtual machine for servers. *ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS'99)*, May 1999, http://www.research.ibm.com/jalapeno/publication.html.

[AD97] Ole Agesen and David Detlefs. Finding references in Java stacks. In Peter Dickman and Paul R. Wilson, editors, *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, October 1997, ftp://ftp.dcs.gla.ac.uk/pub/drastic/gc/detlefs.ps.

[ADH+00] Gerald Aigner, Amer Diwan, David L. Heine, Monica S. Lam, David L. Moore, Brian R. Murphy, and Constantine Sapuntzakis. An overview of the SUIF2 compiler infrastructure, 2000, http://suif.stanford.edu.

[ADM98] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 269–279, Montreal, Canada, 17–19 June 1998.

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988, http://doi.acm.org/10.1145/989393.989417.

[AG98] Alexander Aiken and David Gay. Memory management with explicit regions. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press, http://doi.acm.org/10.1145/277650.277748.

[Age98] Ole Agesen. GC points in a threaded environment. Technical Report SMLI-TR-98-70, Sun Microsystems Laboratories, December 1998, http://research.sun.com/techrep/1998/abstract-70.html.

[AJ99] Malcolm P. Atkinson and Mick J. Jordan. Issues raised by three years of developing PJama: An orthogonally persistent platform for Java. In Catriel Beeri and Peter Buneman, editors, *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings*, volume 1540 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 1999, http://link.springer.de/link/service/series/0558/bibs/1540/15400001.htm.

[AM92] A. Albano and R. Morrison, editors. *Persistent Object Systems: Implementation and Use (Proceedings of the Fifth International Workshop on Persistent Object Systems)*, Workshops in Computing, San Miniato, Italy, September 1992. Springer-Verlag.

[AM95] M. P. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4(3):319–401, 1995, http://www.dcs.st-and.ac.uk/rsch/publications/AM95.shtml.

[AMB95] Malcolm P. Atkinson, David Maier, and Véronique Benzaken, editors. *Persistent Object Systems, Proceedings of the Sixth International Workshop on Persistent Object Systems, Tarascon, Provence, France, 5-9 September 1994*, Workshops in Computing. Springer and British Computer Society, 1995.

[AP99] Gabriel Antoniu and Christian Perez. Using preemptive thread migration to load-balance data-parallel applications. Research Report RR1999-45, LIP, ENS Lyon, France, September 1999, ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR1999/RR1999-45.ps.Z.

[App89] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, February 1989, http://www.cs.princeton.edu/faculty/appel/papers/143.ps.

[Arm98] E. Armstrong. Cover story: HotSpot: A new breed of virtual machine. *JavaWorld: IDG's magazine for the Java community*, 3(3), March 1998, http://www.javaworld.com/javaworld/jw-03-1998/jw-03-hotspot.htm.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bar88] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC Western Research Laboratory, Palo Alto, CA, February 1988, http://www.research.digital.com/wrl/techreports/88.2.ps. Also in Lisp Pointers 1, 6 (April–June 1988), 2–12.

[BB] Joshua Bloch and Gilad Bracha. JSR 201: Extending the Java programming language with enumerations, autoboxing, enhanced for loops and static import, http://jcp.org/en/jsr/detail?id=201.

[BC92] Hans-Juergen Boehm and David R. Chase. A proposal for garbage-collector-safe C compilation. *Journal of C Language Translation*, pages 126–141, 1992, http://reality.sgi.com/employees/boehm_mti/papers/boecha.ps.gz.

[BC99] Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 104–117, Atlanta, May 1999. ACM Press, http://doi.acm.org/10.1145/301618.301648.

[BCF+99] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Java Grande Conference*, pages 129–141, June 1999.

[BD94] Manuel E. Benitez and Jack W. Davidson. Target-specific global code improvement: Principles and applications. Technical Report CS-94-42, Department of Computer Science, University of Virginia, April 1994, http://www.cs.virginia.edu/zephyr/papers.html.

[BFH+92] Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS(R) nanokernel architecture. In USENIX Association, editor, *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures: 27–28 April, 1992, Seattle, WA, USA*, pages 95–112, Berkeley, CA, USA, April 1992. USENIX, http://www.cis.upenn.edu/~KeyKOS/.

[BH00] S. Bouchenak and D. Hagimont. Approaches to capturing Java threads state, 2000, http://sirac.inrialpes.fr/Biblio/publi.html.

[BHL98] Andrew Bernard, Robert Harper, and Peter Lee. How generic is a generic back end? Using MLRISC as a back end for the TIL compiler. In X. Leroy and A. Ohori, editors, *Proceedings of the Workshop on Types in Compilation*, pages 53–77, Kyoto, Japan, March 1998. Springer-Verlag LNCS 1473, http://www-2.cs.cmu.edu/~rwh.

[BHNP98] Luc Bougé, Phil Hatcher, Raymond Namyst, and Christian Perez. A multithreaded runtime environment with thread migration for a HPF data-parallel compiler. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 418–425, Paris, France, October 12–18, 1998. IEEE Computer Society Press.

[BJMM02] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: getting around garbage collection gridlock. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI-02)*, volume 37, 5 of *ACM SIGPLAN Notices*, pages 153–164, New York, June 17–19 2002. ACM Press, http://doi.acm.org/10.1145/512529.512548.

[Bla99] Bruno Blanchet. Escape analysis for object-oriented languages: application to java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 20–34. ACM Press, 1999, http://doi.acm.org/10.1145/320384.320387.

[BLNR96] Eva Z. Bem, Anders Linström, Stephen Norris, and John Rosenberg. Hoppix - an implementation of a Unix server on a persistent operating system. In Luis-Felipe Cabrera and Nayeem Islam, editors, *Proceedings of 5th International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, pages 112–116, Washington, DC, 1996. IEEE Computer Society, http://csdl.computer.org/comp/proceedings/iwooos/1996/7692/00/76920112abs.htm.

[BM02] Stephen M. Blackburn and Kathryn S. McKinley. In or out? putting write barriers in their place. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 175–

184, Berlin, June 2002. ACM Press, http://doi.acm.org/10.1145/773039.512452.

[Boe93] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, pages 197–206, Albuquerque, NM, June 1993. ACM Press, http://doi.acm.org/10.1145/155090.155109.

[Boe96] Hans-Juergen Boehm. Simple garbage-collector-safety. *ACM SIGPLAN Notices*, 31(5):89–98, May 1996, http://doi.acm.org/10.1145/231379.231394. Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).

[Bor] Borland. Turbo Pascal 7.0 website, http://info.borland.com/pascal/tp7fact.html.

[Bou99] S. Bouchenak. Pickling threads state in the Java system, April 1999, citeseer.nj.nec.com/bouchenak99pickling.html. Research Seminar on Advances in Distributed Systems (ERSADS'99).

[BPK94] Micah Beck, James S. Plank, and Gerry Kingsley. Compiler-assisted checkpointing. Technical Report UT-CS-94-269, Department of Computer Science, University of Tennessee, December 1994, ftp://cs.utk.edu/pub/TechReports/1994/ut-cs-94-269.ps.Z.

[Bre88] T. M. Breuel. Lexical closures for C++. In *USENIX Proceedings. C++ Conference*, pages 293–304, 1988, http://people.debian.org/~aaronl/Usenix88-lexic.pdf.

[Bri75] Dianne Ellen Britton. Heap storage management for the programming language Pascal. Master's thesis, University of Arizona, 1975, http://www.druseikis.com/dbritton/msthesis/.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988, http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html.

[Can04] Michaël Van Canneyt. *Reference guide for Free Pascal, version 1.9.2*, January 2004, http://www.freepascal.org/.

[CB01] Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36.5 of *ACM SIGPLAN Notices*, pages 125–136, N.Y., June 20–22 2001. ACMPress, http://doi.acm.org/10.1145/378795.378823.

[CFL93] Jeff Chase, Mike Feeley, and Hank Levy. Some issues for single address space systems. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 150–154, 1993, http://www.cs.washington.edu/homes/levy/opal/opal.html.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19. ACM Press, 1999, http://doi.acm.org/10.1145/320384.320386.

[Cha94] David Chase. Implementation of exception handling, Part II: Calling conventions, asynchrony, optimizers, and debuggers. *The Journal of C Language Translation*, 6(1):20–32, September 1994. (Not reviewed).

[Chi95] Derek Chiou. Using GCC as an efficient, portable back-end, 1995, http://csg-www.lcs.mit.edu:8001/~derek/Student95a.ps. The Proceedings of the MIT Student Workshop for Scalable Computing.

[CHL98] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press, http://doi.acm.org/10.1145/277650.277718.

[CHM97] David Cronk, Matthew Haines, and Piyush Mehrotra. Thread migration in the presence of pointers. In H. El-Rewini and Y. N. Patt, editors, *Proc. of the 30th Hawaii Int'l Conf. on Systems Sciences - HICCS'97*, pages 292–298. IEEE Computer Society Press, January 1997, http://csdl.computer.org/comp/proceedings/hicss/1997/7734/01/7734010292abs.htm.

[CK98] J.-F. Collard and J. Knoop. A comparative study of reaching-definitions analyses. Technical Report 1998/22, University of Versailles, France, 1998, ftp://ftp.par.univie.ac.at/pub/papers.

[CLBHL93] Jeff Chase, Hank Levy, Miche Baker-Harvey, and Ed Lazowska. Opal: A single address space system for 64-bit architectures. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 80–85, 1993, http://www.cs.washington.edu/homes/levy/opal/opal.html.

[DAK00] Robert Dimpsey, Rajiv Arora, and Kean Kuiper. Java server performance: A case study of building efficient, scalable JVMs. *IBM Systems Journal*, 39(1):151–174, 2000, http://www.research.ibm.com/journal/sj/391/dimpsaut.html.

[Day00] L. Daynès. Implementation of automated fine-granularity locking in a persistent programming language. *Software: Practice and Experience*, 30(4):325–361, April 2000, http://www3.interscience.wiley.com/cgi-bin/abstract/71004139/ABSTRACT.

[DCI+97] Andrew Duncan, Bogdan Cocosel, Costin Iancu, Holger Kienle, Radu Rugina, Urs Hölzle, and Martin Rinard. OSUIF: SUIF 2.0 with objects. In *Proceedings of the Second SUIF Compiler Workshop*, Stanford University, August21–23 1997. http://suif.stanford.edu/suifconf/suifconf2/.

[DdBF+94] Alan Dearle, Rex di Bona, James Farrow, Frans Kenskens, Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, Summer 1994, http://docs.dcs.napier.ac.uk/DOCS/GET/dearle94c/document.html.

[DdBF+96] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, David Hulse, Anders Lindstrom, Stephen Norris, John Rosenberg, and Francis Vaughan. Protection in the Grasshopper Operating System. In *Proceedings of the 6th International Workshop on Persistent Object Systems*, Tarascon, France, September 1996. http://docs.dcs.napier.ac.uk/DOCS/GET/dearle94a/document.html.

[DE93] David L. Detlefs and John R. Ellis. Safe, efficient garbage collection for C++. Technical Report 102, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, June 1993. Order from src-report@src.dec.com.

[DGVZ98] A. Dold, T. Gaul, V. Vialard, and W. Zimmermann. ASM-Based Mechanized Verification of Compiler Backends. In Uwe Glässer and Peter H. Schmitt, editors, *Proceedings of the 5th International Workshop on Abstract State Machines*, pages 50–67, Magdeburg, Germany, September 1998. http://i44www.info.uni-karlsruhe.de/~verifix/pres/paper/ASM-WS98-DGVZ.ps.gz.

[DH95] A. Dearle and D. Hulse. On page-based optimistic process checkpointing. In *Proc. of the Fourth Int'l Workshop on Object Orientation in Operating Systems (IWOOOS'95)*, pages 24–32, August 1995, http://docs.dcs.napier.ac.uk/DOCS/GET/dearle95a/document.html.

[Diw91] Amer Diwan. Stack tracing in A statically typed language. In Paul R. Wilson and Barry Hayes, editors, *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, Addendum to OOPSLA'91 Proceedings*, October 1991, ftp://ftp.cs.utexas.edu/pub/garbage/GC91/.

[Diw94] Amer Diwan. Master's project report, January 1994. Department of Computer and Information Science, University of Massachusetts.

[DKL+02] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for Java. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 76–87, Berlin, June 2002. ACM Press, http://www.cs.technion.ac.il/~erez/Papers/TLH-ISMM-02.ps.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978, http://doi.acm.org/10.1145/359642.359655.

[DMH92] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, number 27(7) in SIGPLAN Notices, pages 273–282, San Francisco, CA (USA), June 1992. ACM SIGPLAN, http://doi.acm.org/10.1145/143095.143140.

[DRH+92] Alan Dearle, John Rosenberg, Frans Henskens, Francis Vaughan, and Kevin Maciunas. An examination of operating system support for persistent object systems. In *Proceedings of the Twenty-Fifth Annual Hawaii International Conference on System Sciences*, pages 779–789, 1992, http://docs.dcs.napier.ac.uk/DOCS/GET/dearle92a/document.ps.gz.

[DWA93] *DWARF Debugging Information Format, revision 2.0.0*, July 1993, http://www.eagercon.com/dwarf/dwarf3std.htm.

[Eng00] Ralf S. Engelschall. Portable multithreading: The signal stack trick for user-space thread creation. In *Proceedings of the USENIX Annual Conference*, pages 239–250, San Diego, California, USA, June 2000. USENIX Association, http://www.usenix.org/events/usenix2000/general/engelschall.html.

[Eto03] Hiroaki Etoh. GCC extension for protecting applications from stack-smashing attacks (ProPolice), 2003, http://www.trl.ibm.com/projects/security/ssp/. IBM Research.

[FK97] Michael Franz and Thomas Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, December 1997, http://www.ics.uci.edu/~franz/SlimBinaries.html.

[Fla97] D. Flanagan. *Java In A Nutshell*. A Nutshell Handbook. O'Reilly, 2nd edition, 1997.

[FR] Kathleen Fisher and John Reppy. The MOBY programming language website, http://moby.cs.uchicago.edu/. Department of Computer and Information Science, University of Massachusetts.

[FR02] Kathleen Fisher and John Reppy. Compiler support for lightweight concurrency, http://moby.cs.uchicago.edu/papers/. Submitted for publication, 2002.

[Fra91] Christopher W. Fraser. A retargetable compiler for ANSI C. *ACM SIGPLAN Notices*, 26(10):29–43, 1991, http://doi.acm.org/10.1145/122616.122621.

[Gan94] Ravichandran Ganesan. Local variable allocation for accurate garbage collection of C++. Master's thesis, Iowa State University, July 1994, http://archives.cs.

iastate.edu/documents/disk0/00/00/00/78/index.html. Technical report ISUTR 94–12.

[GJ86] Carlo Ghezzi and Mehdi Jazayeri. *Programming language concepts (2nd ed.)*. John Wiley & Sons, Inc., 1986.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000, http://citeseer.nj.nec.com/gosling00java.html.

[GL03] Lal George and Allen Leung. MLRISC: A framework for retargetable and optimizing compiler back ends, January 2003, http://www.cs.nyu.edu/leunga/www/MLRISC/Doc/html/index.html.

[GM00] Dan Grossman and Greg Morrisett. Scalable certification for typed assembly language. In *Workshop on Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 117–145, Montreal, Canada, September 2000. Springer-Verlag.

[GS98] David Gay and Bjarne Steensgaard. Stack allocating objects in Java. Technical report, Microsoft Research, October 1998, http://research.microsoft.com/apl/stackalloc-abstract.pdf.

[GS00] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction (CC'2000)*, volume 1781 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000, http://citeseer.nj.nec.com/gay00fast.html.

[GS04] John Gilmore and Stan Shebs. *GDB Internals Manual*. Cygnus Solutions, February 2004, http://www.gnu.org/software/gdb/documentation/. Second edition.

[Har01] Timothy L Harris. Dynamic adaptive pre-tenuring. In *International Symposium on Memory Management (ISMM '00)*, volume 36(1), pages 127–136, 2001, http://citeseer.nj.nec.com/article/harris00dynamic.html.

[HB87] D. M. Harland and B. Beloff. OBJEKT: A persistent object store with an integrated garbage collector. *ACM SIGPLAN Notices*, 22(4):70–79, April 1987, http://doi.acm.org/10.1145/24714.24723.

[HC99a] Antony L. Hosking and Jiawan Chen. Mostly-copying reachability-based orthogonal persistence. *ACM SIGPLAN Notices*, 34(10):382–398, 1999, http://doi.acm.org/10.1145/320384.320427.

[HC99b] Antony L. Hosking and Jiawan Chen. PM3: An orthogonal persistent systems programming language — design, implementation, performance. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *Proceedings of the Twenty-fifth International Conference on Very*

Large Databases, Edinburgh, Scotland, UK, 7–10 September, 1999, pages 587–598, Los Altos, CA 94022, USA, 1999. Morgan Kaufmann Publishers, http://www.vldb.org/dblp/db/conf/vldb/HoskingC99.html.

[HD96] David Hulse and Alan Dearle. A log-structured persistent store. In *Proceedings of the 19th Australasian Computer Science Conference*, 1996, http://docs.dcs.napier.ac.uk/DOCS/GET/hulse96a/document.html.

[HD00] Martin Hirzel and Amer Diwan. On the type accuracy of garbage collection. In Tony Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press, http://doi.acm.org/10.1145/362422.362428.

[HDH01] Martin Hirzel, Amer Diwan, and Antony L. Hosking. On the usefulness of liveness for garbage collection and leak detection. In J. Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object Oriented Programming*, volume 2072 / 2001, pages 181–206, Budapest, Hungary, June 2001. Springer-Verlag, http://www.cs.colorado.edu/~diwan/ecoop01-gc-liveness.pdf.

[HDH02] Martin Hirzel, Amer Diwan, and Johannes Henkel. On the usefulness of type and liveness for garbage collection and leak detection. *ACM Transactions on Programming Languages and Systems*, 24(6):593–624, November 2002, http://doi.acm.org/10.1145/586088.586089.

[Hen02] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 150–156, Berlin, June 2002. ACM Press, http://doi.acm.org/10.1145/512429.512449.

[HIB+02] Teresa Higuera, Valerie Issarny, Michel Banatre, Gilbert Cabillic, Jean-Philippe Lesot, and Frederic Parain. Memory management for real-time Java: an efficient solution using hardware support. *Real-Time Systems Journal*, 2002, http://www-rocq.inria.fr/arles/doc/doc.html.

[Hir00] Martin Hirzel. Effectiveness of garbage collection and explicit deallocation. Master's thesis, University of Colorado, 2000, http://www-plan.cs.colorado.edu/hirzel/papers/.

[HM90] Antony L. Hosking and J. Eliot B. Moss. Towards compile-time optimisations for persistence. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Proceedings of the International Workshop on Persistent Object Systems*, Implementing Persistent Object Bases: Principles and Practice, pages 17–27, Martha's Vineyard, Massachusetts, September 1990. Morgan Kaufmann, ftp://ftp.cs.umass.edu/pub/osl/papers/pos90.ps.Z.

[HM93] Antony L. Hosking and J. Eliot B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In Barbara Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 106–119, New York, NY, USA, December 1993. ACM Press.

[HM95] Antony L. Hosking and J. Eliot B. Moss. Lightweight write detection and checkpointing for fine-grained persistence. Technical Report 95-084, Purdue University, 1995, ftp://ftp.cs.purdue.edu/pub/hosking/papers/tods.ps.gz.

[HM01] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM Java Grande — ISCOPE 2001 Conference*, Stanford University, CA, 2001. http://doi.acm.org/10.1145/376656.376810.

[HMDW91] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. COINS Technical Report 91-47, University of Massachusetts, Object Oriented Systems Laboratory, Department of comp. and Info. Science, Amherst, MA, 01003, September 1991, ftp://ftp.cs.umass.edu/pub/osl/papers/tr9147.ps.Z. Only available online.

[HMMM97] Richard L. Hudson, Ron Morrison, J. Eliot B. Moss, and David S. Munro. Garbage collecting the world: One car at a time. In *OOPSLA'97 ACM Conference on Object-Oriented Systems, Languages and Applications — Twelth Annual Conference*, volume 32(10) of *ACM SIGPLAN Notices*, Atlanta, GA, October 1997. ACM Press, http://doi.acm.org/10.1145/263700.264353.

[HMMM98] Richard L. Hudson, Ron Morrison, J. Eliot B. Moss, and David S. Munro. Where have all the pointers gone? In *Proceedings of 21st Australasian Computer Science Conference*, pages 107–119, Perth, 1998. http://www-ppg.dcs.st-and.ac.uk/Publications/PostScript/dmos.pointers.ps.gz.

[HMS92] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, October 1992. http://www.cs.purdue.edu/homes/hosking/papers.html.

[HMSW00] Richard L. Hudson, J. Eliot B. Moss, Sreenivas Subramoney, and Weldon Washburn. Cycles to recycle: Garbage collection on the IA-64. In Tony Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press.

[HN97] Antony L. Hosking and Aria P. Novianto. Reachability-based orthogonal persistence for C, C++ and other intransigents. In Peter Dickman and Paul R. Wilson, editors, *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, October 1997, http://www.cs.purdue.edu/homes/hosking/papers.html.

[HNB99] Brent Hailpern, Linda M. Northrop, and A. Michael Berman. Proceedings of the 14th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, 1999, http://portal.acm.org/toc.cfm?id=320384.

[HNCB98] Antony L. Hosking, Nathaniel Nystrom, Quintin Cutts, and Kumar Brahnmath. Optimizing the read and write barrier for orthogonal persistence. In *Proceedings of the Eighth International Workshop on Persistent Object Systems*, Tiburon, CA, August 1998. http://www.cs.purdue.edu/homes/hosking/papers.html.

[Hos91] Anthony L. Hosking. Main memory management for persistence. In Paul R. Wilson and Barry Hayes, editors, *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, Addendum to OOPSLA'91 Proceedings*, October 1991, http://www.cs.purdue.edu/homes/hosking/papers.html.

[Hos95] Antony L. Hosking. *Lightweight support for fine-grained persistence on stock hardware*. PhD thesis, University of Massachusetts at Amherst, February 1995, http://www.cs.purdue.edu/homes/hosking/papers.html.

[Hul96] David Hulse. A flexible persistent architecture permitting trade-off between snapshot and recovery times. Technical Report GH-16, University of Sydney, Computer Science, N.S.W 2006, Australia, 1996, http://docs.dcs.napier.ac.uk/DOCS/GET/hulse96b/document.html.

[HVER97] Gernot Heiser, Jerry Vochteloo, Kevin Elphinstone, and Stephen Russell. The Mungi kernel API, release 1.0. Technical Report UNSW-CSE-TR-9701, University of New South Wales, Department of Computer Systems, April 1997, ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/9701.ps.Z.

[Int90a] International Organization for Standardization. *Extended Pascal ISO 10206:1990*. Geneva, Switzerland, 1990, http://pascal-central.com/standards.html.

[Int90b] International Organization for Standardization. *Pascal Standard ISO 7185:1990*. Geneva, Switzerland, 1990, http://pascal-central.com/standards.html.

[Int01] Intel Corporation, Santa Clara, CA. *IA-32 Intel Architecture Software Developers Manual*, 2001, http://developer.intel.com/design/pentiumii/manuals/. Order numbers: 245470, 245471, 245472.

[Jon96] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996, http://www.cs.ukc.ac.uk/people/staff/rej/gcbook/gcbook.html. With a chapter on Distributed Garbage Collection by R. Lins.

[JR98] Simon L. Peyton Jones and Norman Ramsey. Machine-independent support for garbage collection, debugging, exception handling and concurrency. Technical Report CS–98–19, University of Virginia, August 1998, http://www.eecs.harvard.edu/~nr/pubs/c--rti-abstract.html.

[JRR99] S. L. Peyton Jones, N. Ramsey, and F. Reig. C--: a Portable Assembly Language that Supports Garbage Collection. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99), Paris, France*, volume 1702 of *Lecture Notes in Computer Science*, pages 1–28. Springer-Verlag, Berlin, Germany, 1999, http://www.cminusminus.org/abstracts/ppdp.html.

[KA99] Prasad Kakulavarapu and José Nelson Amaral. A survey of load balancers in modern multi-threading systems. In *Proceedings of the 11th Symposium on Computer Architecture and High Performance Computing*, pages 10–16, Natal, Brazil,, September 1999. http://www.cs.ualberta.ca/~amaral/papers/mtsurvey_sbac99.ps.gz.

[Kak98] Sheetal V. Kakkad. Address translation and storage management for persistent object stores. Technical Report CS-TR-98-07, University of Texas, Austin, March 1, 1998, ftp://ftp.cs.utexas.edu/pub/techreports/tr98-07.ps.Z.

[Kak99] Kamala Prasad Kakulavarapu. Dynamic load balancing issues in the earth runtime system. Master's thesis, School of Computer Science McGill University, Montréal Québec, Canada, December 1999.

[KCC+97] G. N. C. Kirby, R. C. H. Connor, Q. I. Cutts, R. Morrison, D. S. Munro, and S. Scheuerl. Flask: An architecture supporting concurrent distributed persistent applications. Technical Report CS/97/4, University of St Andrews, Scotland, 1997.

[KF99] Thomas Kistler and Michael Franz. A tree-based alternative to Java byte-codes. *International Journal of Parallel Programming*, 27(1):21–33, 1999.

[KF00] Thomas Kistler and Michael Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems*, 22(3):490–505, 2000, http://doi.acm.org/10.1145/353926.353937.

[Kin03] Andy C. King. Removing GC synchronisation (extended version). Technical Report 11-03, University of Kent, April 2003, http://www.cs.kent.ac.uk/pubs/2003/1614. Winner (Graduate Division) ACM Student Research Competition.

[KJW98] Sheetal V. Kakkad, Mark S. Johnstone, and Paul R. Wilson. Portable run-time type description for conventional compilers. In Richard Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 146–153, Vancouver, October 1998. ACM Press, http://doi.acm.org/10.1145/301589.286876.

[KKR+86] David A. Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, volume 21(7) of

ACM SIGPLAN Notices, pages 219–233, Palo Alto, CA, June 1986. ACM Press, http://doi.acm.org/10.1145/13310.13333.

[KM97] G. N. C. Kirby and R. Morrison. Orthogonal persistence as an implementation platform for software development environments. Technical Report CS/97/6, University of St Andrews, 1997, http://www-ppg.dcs.st-and.ac.uk/Publications/1997.html.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Lan92] Charles R. Landau. The checkpoint mechanism in KeyKOS. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 86–91. IEEE Computer Society, September 1992, http://ieeexplore.ieee.org/xpl/abs_free.jsp?arNumber=252995.

[LDdB+94] Anders Lindstrom, Alan Dearle, Rex di Bona, J. Matthew Farrow, Frans Henskens, John Rosenberg, and Francis Vaughan. A model for user-level memory management in a persistent distributed environment. In Gopal Gupta, editor, *Proceedings of the Seventeenth Annual Computer Science Conference, ACSC-17, Part B*, pages 343–354, Christchurch, New Zealand, January 1994. http://docs.dcs.napier.ac.uk/DOCS/GET/lindstrom94a/document.html.

[LDdB+95] Anders Lindstrom, Alan Dearle, Rex di Bona, Stephen Norris, John Rosenberg, and Francis Vaughan. Persistence in the Grasshopper kernel. In Ramamohanarao Kotagiri, editor, *Proceedings of the Eighteenth Australasian Computer Science Conference, ACSC-18*, pages 329–338, Glenelg, South Australia, February 1995. IEEE Computer Society, http://docs.dcs.napier.ac.uk/DOCS/GET/lindstrom95a/document.html.

[Lea99] Doug Lea. *Concurrent Programming in Java™ Second Edition: Design principles and Patterns*. The Java Series. Addison-Wesley, 2nd edition, 1999.

[LRD95] Anders Lindstrom, John Rosenberg, and Alan Dearle. The grand unified theory of address spaces. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 66–71, Orcas Island, Washington, May 1995. http://docs.dcs.napier.ac.uk/DOCS/GET/lindstrom95b/document.html.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, second edition, 1999, http://java.sun.com/docs/books/vmspec/.

[MBC+96] R. Morrison, A. L. Brown, R. C. H. Connor, Q. I. Cutts, A. Dearle, G. N. C. Kirby, and D. S. Munro. The Napier88 reference manual (release 2.2.1). Technical report, University of St Andrews, 1996, http://www.dcs.st-and.ac.uk/research/publications/MBC+96b.php.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Informal Proceedings of the Workshop on Compiler Support for Systems Software*, Atlanta, Georgia, May 1999. http://www.cs.cornell.edu/talc/.

[MDB+85] R. Morrison, A. Dearle, P. J. Bailey, A. L. Brown, and M. P. Atkinson. The persistent store as an enabling technology for integrated project support environments. In *Proceedings of the 8th International Conference on Software Engineering*, pages 166–173. IEEE Computer Society Press, August 1985.

[Met] Metrowerks Inc. *CodeWarrior Pascal: Language Reference*, http://wwwpa.win.tue.nl/facilities/metrowerks/cw/updates/Pascal_Language_Ref.pdf.

[MH94] J. Eliot B. Moss and Antony L. Hosking. Expressing object residency optimizations using pointer type annotations. In Malcolm Atkinson, David Maier, and Véronique Benzaken, editors, *Proceedings of the International Workshop on Persistent Object Systems*, Workshops in Computing, pages 3–15, Tarascon, France, September 1994. Springer-Verlag, 1995, ftp://ftp.cs.umass.edu/pub/osl/papers/pos94.ps.Z.

[MHL+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992, http://icg.harvard.edu/~cs265/lectures/mohan-1992.pdf.

[MK87] J. Eliot B. Moss and Walter H. Kohler. Concurrency features for the Trellis/Owl language. In *European Conference on Object-Oriented Programming*, pages 223–232, Paris, France, 1987.

[MKM] Julia Menapace, Jim Kingdon, and David MacKenzie. *The "stabs" debug format*, http://docs.freebsd.org/info/stabs/stabs.pdf. Revision 2.128.

[MMS95] Bernd Mathiske, Florian Matthes, and Joachim W. Schmidt. On migrating threads. In *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Naharia, Israel*, June 1995, http://www.sts.tu-harburg.de/papers/1995/MMS95a. Also appeared as TR FIDE/95/136 , FIDE Technical Report Series, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ.

[Mot92] Motorola Inc. *M68000 Family Programmer's Reference Manual*, 1992, http://e-www.motorola.com/collateral/M68000PRM.pdf. (Ref: M68000PM/AD).

[Mot93] Motorola Inc. *PowerPC^{TM} 601 RISC Microprocessor User's Manual*, 1993, http://e-www.motorola.com/files/32bit/doc/user_guide/MPC601UM.pdf. (Ref: MPC601UM/AD).

[MR96] Edward Mascarenhas and Vernon Rego. Ariadne: Architecture of a portable threads system supporting thread migration. *Software Practice and Experience*, 26(3):327–356, March 1996, http://www3.interscience.wiley.com/cgi-bin/abstract?ID=16793.

[MS94] Florian Matthes and Joachim W. Schmidt. Persistent threads. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 403–414. Morgan Kaufmann, 1994, http://www.vldb.org/conf/1994/P403.PDF.

[Mue97] F. Mueller. Distributed shared memory threads: DSM-threads. In *Proc. of the Workshop on Run-Time Systems for Parallel Programming*, pages 31–40, April 1997, http://citeseer.nj.nec.com/mueller97distributed.html.

[Mut97] Robert Muth. Register liveness analysis of executable code, November 1997, http://www.cs.arizona.edu/alto. Department of Computer Science, The University of Arizona.

[mWHC92] Wen mei W. Hwu and Pohua P. Chang. Efficient instruction sequencing with inline target insertion. *IEEE Transactions on Computers*, 41(12):1537–1551, 1992, http://citeseer.ist.psu.edu/hwu90efficient.html.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[NO93] Scott Nettles and James O'Toole. Real-time replication garbage collection. In Robert Cartwright, editor, *Proceedings of the Conference on Programming Language Design and Implementation*, pages 217–226, New York, NY, USA, June 1993. ACM Press, http://doi.acm.org/10.1145/155090.155111.

[OHL99] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 303–313, Newport Beach, California, October 12–16, 1999. IEEE Computer Society Press, http://csdl.computer.org/comp/proceedings/pact/1999/0425/00/04250303abs.htm.

[Ope03] The IEEE and The Open Group. *The Open Group Base Specifications Issue 6*, 2003, http://www.opengroup.org/onlinepubs/007904975/. IEEE Std 1003.1, 2003 Edition.

[OW99] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, second edition, 1999, http://www.oreilly.com/catalog/jthreads2/.

[PG02] Tony Printezis and Alex Garthwaite. Visualising the Train garbage collector. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 100–105, Berlin, June 2002. ACM Press, http://doi.acm.org/10.1145/512429.512436.

[Pie01] Matt Pietrek. Under the hood: IA-64 registers, part 2. *MSDN Magazine*, 7(16), July 2001, http://msdn.microsoft.com/msdnmag/issues/01/07/hood.

[Piz97] Markus Pizka. Design and implementation of the GNU INSEL compiler gic. Technical Report TUM–I 9713, Technische Universität München, Institut für Informatik, 1997, http://wwwbroy.informatik.tu-muenchen.de/~pizka/.

[Piz99] Markus Pizka. Thread segment stacks. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - PDPTA'99*, June 1999, http://wwwbroy.informatik.tu-muenchen.de/~pizka/pdpta99.final.ps.

[Pri00] Tony Printezis. *Management of Long-Running High-Performance Persistent Object Stores*. PhD thesis, Department of Computing Science, University of Glasgow, Scotland, May 2000.

[Pro95] Todd A. Proebsting. BURS automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, May 1995, http://doi.acm.org/10.1145/203095.203098.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, September 1999, http://www.acm.org/pubs/citations/journals/toplas/1999-21-5/p895-poletto/.

[RDH+96] John Rosenberg, Alan Dearle, David Hulse, Anders Lindström, and Stephen Norris. Operating system support for persistent and recoverable computations. *Communications of the ACM*, 39(9):62–69, September 1996, http://www.acm.org/pubs/contents/journals/cacm/1996-39/.

[RI82] Jonathan A. Rees and Norman I. Adams IV. T: a dialect of Lisp or LAMBDA: The ultimate software tool. In *Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 114–122. ACM Press, 1982, http://portal.acm.org/citation.cfm?id=802142.

[RJ96] Helena C. C. D. Rodrigues and Richard E. Jones. A cyclic distributed garbage collector for Network Objects. In Ozalp Babaoglu and Keith Marzullo, editors, *Tenth International Workshop on Distributed Algorithms WDAG'96*, number 1151, pages 123–140, Bologna, Italy, October 1996. Springer, http://www.cs.kent.ac.uk/pubs/1996/12/.

[RJ00] Norman Ramsey and Simon Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 285–298. ACM Press, 2000, http://doi.acm.org/10.1145/349299.349337.

[Ruf00] Erik Ruf. Removing synchronization operations from Java. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Vancouver, June 2000. ACM Press, http://research.microsoft.com/~ruf/preprint.htm.

[Sal01] Alexandru Salcianu. Pointer analysis and its applications for Java programs. Master's thesis, Massachusetts Institute of Technology, September 2001, http://www.mit.edu/people/salcianu/publications/sm-thesis.ps.

[San90] The Santa Cruz Operation, Inc. *System V Application Binary Interface: SPARC processor supplement*, third edition, 1990, http://www.sparc.com/standards/psABI3rd.pdf.

[Sat94] S. Satishkumar. Register allocation for accurate garbage collection of C++. Master's thesis, Iowa State University, July 1994, http://www.cs.iastate.edu/tech-reports/TR94-13.ps. Technical report ISUTR 94–12.

[SCM99] Olin Shivers, James W. Clark, and Roland McGrath. Atomic heap transactions and fine-grain interrupts. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP-99)*, volume 34.9 of *ACM Sigplan Notices*, pages 48–59, N.Y., September 27–29 1999. ACM Press.

[SF99] Naoya Suzuki and Munehiro Fukuda. A design of self-migrating threads in C++. Technical Report ISE-TR-99-160, Institute of Information Sciences and Electronics, University of Tsukuba, May 1999, http://iris.is.tsukuba.ac.jp/~fukuda/.

[SFS96] Jonathan S. Shapiro, David J. Farber, and Jonathan M. Smith. State caching in the EROS kernel. In *Proceedings of the 7th International Workshop on Persistent Object Systems*, Cape May, N.J., 1996. http://www.eros-os.org/papers/pos96.ps.

[SH96] Peter Smith and Norman C. Hutchinson. Heterogeneous process migration: The Tui system. Technical Report TR-96-04, Department of Computer Science, University of British Columbia, February 1996, ftp://ftp.cs.ubc.ca/pub/local/techreports/1996/TR-96-04.ps.gz. Tue, 22 Jul 1997 22:20:09 GMT.

[SHB+02] Darko Stefanović, Matthew Hertz, Stephen M. Blackburn, Kathryn S. McKinley, and J. Eliot B. Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In Loren Meissner, editor, *ACM SIGPLAN Workshop on Memory*

System Performance (MSP), ACM Sigplan Notices, Berlin, Germany, June16 2002. http://doi.acm.org/10.1145/773039.773042.

[She03] Tzu-Yung Shen. Assume-guarantee based formal verification of hierarchical software designs. Master's thesis, Embedded System Laboratory at Computer Science and Information Engineering, National Chung Cheng University, 160 San-Hsing, Min-Hsiung, Chia-Yi 621 Taiwan, 2003, http://embedded.cs.ccu.edu.tw/paper/Tzu-YungShenThesis2003.pdf.

[SKS00] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. On the effectiveness of GC in Java. In Tony Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press, http://citeseer.nj.nec.com/shaham00effectiveness.html.

[SKW92] K. Singhal, S. Kakkad, and P. Wilson. Texas: An efficient, portable persistent store. In *Proc. Fifth International Workshop on Persistent Object Systems*, pages 13–28, San Miniato Pisa (Italy), September 1992. ftp://ftp.cs.utexas.edu/pub/garbage/texaspstore.ps.

[SLC99] James M. Stichnoth, Guei-Yuan Lueh, and Michal Cierniak. Support for garbage collection at every instruction in a Java compiler. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 118–127, Atlanta, May 1–4, 1999. ACM Press, http://doi.acm.org/10.1145/301618.301652.

[SM98a] Alan Skousen and Donald Miller. The Sombrero distributed single address space operating system project. In *Proceedings of the 2nd USENIX Windows NT Symposium (WINNT-98)*, pages 168–168, Berkeley, August 3–5 1998. USENIX Association, ftp://ftp.eas.asu.edu/pub/cse/sasos/usenix_nt.pdf.

[SM98b] Frederick Smith and Greg Morrisett. Comparing mostly-copying and mark-sweep conservative collection. In *Proceedings of the First International Symposium on Memory Management*, pages 68–78. ACM Press, 1998, http://www.cs.cornell.edu/talc/papers/mcc-ismm.pdf.

[SM99] Alan Skousen and Donald Miller. Using a single address space operating system for distributed computing and high performance. In *18th IEEE International Performance, Computing, and Communications Conference*, pages 8–14, February 1999, ftp://ftp.eas.asu.edu/pub/cse/sasos/ipccc99.pdf.

[SMM99] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, pages 370–381, N.

Y., November 1–5 1999. ACM Press, http://citeseer.nj.nec.com/123024.html.

[SOT+00] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatso, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000, http://www.research.ibm.com/journal/sj/391/suganuma.html.

[Spa92] SPARC International Inc. *The SPARC Architecture Manual: Version 8*, 1992, http://www.sparc.com/standards/.

[Spa94] SPARC International Inc. *The SPARC Architecture Manual: Version 9*, 1994, http://www.sparc.com/standards/.

[SS92] D. Stein and D. Shah. Implementing lightweight threads. In USENIX Association, editor, *Proceedings of the Summer 1992 USENIX Conference: June 8–12, 1992, San Antonio, Texas, USA*, pages 1–10, Berkeley, CA, USA, Summer 1992. USENIX, http://www.cs.rice.edu/~amsaha/Comp620/Paper4_03rdMar/stein92implementing.pdf.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *17th ACM Symposium on Operating Systems Principles (SOSP '99)*, 1999, http://www.eros-os.org/papers/sosp99-eros-preprint.ps.

[Staa] Richard Stallman. *GNU Compiler Collection Internals - v3.4 - updated 28 December 2002*. Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, http://gcc.gnu.org/onlinedocs/.

[Stab] Richard Stallman. *Using the GNU Compiler Collection - Version 3.3*. Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, http://gcc.gnu.org/onlinedocs/.

[Ste00] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In Tony Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press, http://doi.acm.org/10.1145/362422.362432.

[Sun99] Sun Microsystems. *picoJava-II Programmer's Reference Manual*, March 1999, http://spacejug.org/resources/Embedded_Java/picoJava/picoJava-II.pdf.

[Sun02] Sun Microsystems, 901 San Antonio Road, Palo Alto, CA 94303-4900, USA. *SPARC Assembly Language Reference Manual*, May 2002, http://docs.sun.com/db/doc/816-1681.

[Tar00] David Tarditi. Compact garbage collection tables. In Tony Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press, http://doi.acm.org/10.1145/362422.362437.

[TK99] K. Thitikamol and P. Keleher. Thread migration and communication minimization in DSM systems. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):487–497, March 1999, http://ieeexplore.ieee.org/xpl/abs_free.jsp?arNumber=747869.

[TMG+02] Jim Trevor, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Usenix Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002. http://www.cs.cornell.edu/projects/cyclone/.

[TSKS83] A. S. Tanenbaum, H. Van Staversen, E. G. Keizer, and J. W. Stevenson. A practical tool kit for making portable compilers. *Communications of the Association of Computing Machinery*, 26(9):654–660, September 1983.

[TvRvS+90] Andrew S. Tanenbaum, Robert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experience with the Amoeba distributed operating system. *CACM*, 33(12):46–63, December 1990.

[VD92] Francis Vaughan and Alan Dearle. Supporting large persistent stores using conventional hardware. In *Fifth International Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992. Springer-Verlag, http://docs.dcs.napier.ac.uk/DOCS/GET/vaughan92a/document.ps.gz. available online only, in ftp.gh.cs.su.oz.au as GH-02.

[Voc98] Jerry Vochteloo. *Design, implementation and performance of protection in the Mungi single-address-space operating system*. PhD thesis, University of New South Wales, 1998, ftp://ftp.cse.unsw.edu.au/pub/users/disy/papers/Vochteloo:phd.ps.gz.

[WA00] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors (extended version). Technical Report TR-624-00, Princeton University, Computer Science, December 2000, http://ncstrl.cs.Princeton.EDU/expand.php?id=TR-624-00.

[WG95] W. M. Waite and G. Goos. *Compiler Construction*. Springer, New York, 1995, ftp://i44ftp.info.uni-karlsruhe.de/pub/papers/ggoos/CompilerConstruction.ps.gz.

[WG98] Derek White and Alex Garthwaite. The GC interface in the EVM. Technical Report SML TR–98–67, Sun Microsystems Laboratories, December 1998, http://research.sun.com/research/techrep/1998/abstract-67.html.

[WGQH97] Boris Weissman, Benedict Gomes, Jürgen W. Quittek, and Michael Holtkamp. A performance evaluation of fine grain thread migration with active threads. Technical Report TR-97-054, International Computer Science Institute, Berkeley, CA, December 1997.

[WGQH98] B. Weissman, B. Gomes, J. Quittek, and M. Holtkamp. Efficient fine-grain thread migration with active threads. In *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98)*, pages 410–414, Los Alamitos, March 30–April 3 1998. IEEE Computer Society.

[Wir88] Niklaus Wirth. From Modula to Oberon. *Software Practice and Experience*, 18(7), July 1988, ftp://ftp.inf.ethz.ch/pub/software/Oberon/OberonV4/Docu/ModToOberon.ps.gz. Originally released as ETH CS Dept technical report number 143.

[WK92] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge addresses on standard hardware. In *International Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, September 1992. IEEE Press, ftp://ftp.cs.utexas.edu/pub/garbage/swizz.ps.

[WMR+95] Tim Wilkinson, Kevin Murray, Stephen Russell, Gernot Heiser, and Jochen Liedtke. Single address space operating systems. Technical Report UNSW-CSE-TR-9504, School of Computer Science and Engineering, University of New South Wales, Australia, 1995, ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/9504.ps.Z.

[Wol99] Mario Wolczko. Using a Tracing Java Virtual Machine to gather data on the behavior of Java programs. Technical Report SML 98-0154, Sun Microsystems, March 1999, http://research.sun.com/people/mario/tracing-jvm/.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 187–206. ACM Press, 1999, http://doi.acm.org/10.1145/320384.320400.

[XMR00] Zhichen Xu, Barton P. Miller, and Thomas Reps. Safety checking of machine code. *ACM SIGPLAN Notices*, 35(5):70–82, 2000, http://citeseer.ist.psu.edu/xu00safety.html.