JMTk: A Portable Memory Management Toolkit

Robin John Garner

A subthesis submitted in partial fulfillment of the degree of
Bachelor of Science (Honours) at
The Department of Computer Science
Australian National University

December 2003

Typeset in Palatino by TeX and $\LaTeX\,2_\varepsilon$.

Except where otherwise indicated, this thesis is my own original work.

Robin John Garner
12 December 2003

Acknowledgements

I would like to thank the people who contributed most to this project and thesis:

Steve Blackburn, for ideas, support, inspiration, encouragement, proofreading and (along with Perry and Kathryn) for JMTk.

Andrew Gray, for his part in the porting of JMTk.

Clem Baker-Finch, who introduced me to Haskell—now that this project is finished I can perhaps go back to writing in my language of choice!

And finally to Jeni Allenby without whose support, love and tolerance I could never have completed this degree.

Abstract

JMTk is a memory management toolkit written in Java for the JikesRVM Java Virtual Machine. JMTk provides reusable components for the development of allocation and garbage collection algorithms, and an efficient implementation of a rapidly growing number of memory management schemes, and is a significant enabler for memory management research. A portable version of JMTk would allow the growing body of memory management research done using it to be repeated in different language environments, and to study the different performance characteristics of algorithms across programming languages without the variation in implementation techniques that clouds such issues today.

This thesis describes a project that has ported JMTk for use in C/C++ based language runtimes. The project has used the gcj ahead-of-time java compiler and a source transformation technique to produce a version of JMTk that can efficiently perform memory management for traditionally compiled systems. It is currently running in a testbed environment, where it provides interesting comparisons between accurate and conservative garbage collection techniques. The source transformation tool is currently in use by a team at Purdue, to integrate JMTk into the OVM Java Virtual machine.

This thesis also describes partially completed work to incorporate JMTk into the runtime of the Glasgow Haskell Compiler (ghc), and describes the difficulties that were encountered in the process, and an approach that should enable the work to be completed in the future.

The thesis describes the architecture and construction of the ported system, the source code transformation process used, and evaluates the performance of the final product. It also explores possible future extensions of the techniques used, looks at the issues surrounding working with complex software such as the ghc runtime, and provides a brief evaluation of Java as a system programming language.

The key contribution of this thesis is therefore the availability of JMTk to researchers and developers of language runtimes, and a convincing demonstration of the use of Java in a new application domain.

Contents

Chapter 1

Introduction

Automatic memory management (garbage collection) has been a feature of high level languages since the original LISP system of 1960, and is of increasing importance due to the popularity of modern programming languages such as Java and C#, both of which require an automatically managed heap.

The literature on garbage collection is extensive, but because of the effort involved in building a garbage collector from scratch—in particular the notoriously difficult task of debugging one—most of the literature consists of (at best) A/B comparisons between the author's new algorithm and the preceeding implementation. Empirical knowledge of the relative performance of GC algorithms is sketchy at best, and in particular, little is known about whether performance characteristics exhibited in one language environment are due to implementation details, or are a property of the algorithm in that language environment.

JMTk is a toolkit designed to offer the basic building blocks of memory management, and make it easy to combine them into full featured memory management systems. Its implementation in the Jikes RVM Java virtual machine has made possible for the first time a comparison of all major GC algorithms in the same environment.

The aim of this project is to port JMTk so that the same comparison can be extended to other language runtimes, in particular to the GHC and nhc98 Haskell compiler runtimes.

The approach taken has been to refactor the JMTk codebase to make it more portable; to use automatic source code transformation to adapt JikesRVM-specific Java idioms to a portable form; to use the gcj ahead-of-time java compiler to produce object code that can be efficiently called from a C language environment; and to write a C testbed environment to perform testing and performance tuning on the ported code.

The testbed environment permits a direct comparison between the Boehm-Demers-Wieser conservative garbage collector and accurate collection techniques.

The thesis falls into 3 parts. The first part provides background information to memory management and JMTk, and identifies the problems that the project needs to solve. The second part describes the achievements of the project, and the third part evaluates the resulting system for performance, and draws some conclusions.

This chapter has briefly outlined the project. The next chapter will review the literature on garbage collection, and place the project in context.

Part I

Background

Chapter 2

Background

Automatic memory management has been an active area of research for over 40 years. Changes in computer technology, and in particular the increasing gap in performance between CPU and Memory, makes new ideas practical and old trade-offs unacceptable. This chapter discusses the development of memory management research since its inception, and places this project within the context of current research.

2.1 Memory management

There is a watershed in every beginning programmer's development when they move from simple programs based on arrays and other fixed size data structures, to programs requiring dynamic data structures such as lists, trees etc. One of the difficulties faced by students at this stage[1] is is how to de-allocate these structures when they are no longer needed, and deal with the problems of pointers that remain to data that has been freed (or worse, re-used), track down memory leaks and so on. The most frequently cited estimate is from Rovner [1985][2], which reports that Mesa programmers spent 40% of their development time implementing memory management procedures and finding errors related to explicit storage reclamation.

The solution used by functional programming languages since their inception[3], and adopted by modern programming languages such as Java and C# is to define in the language specification that the heap is automatically managed, i.e. that memory dynamically allocated is automatically reclaimed when it is no longer required. This frees the programmer from mechanical issues of memory management, but at a cost in either program responsiveness, run-time efficiency or both.

In both manually and automatically managed heaps, speed and efficiency of allocation is also an issue. Memory needs to be structured in a way that allows allocation requests to be rapidly completed, especially the re-allocation of reclaimed memory. A

[1] At least programmers trained in the traditional way—students doing their first programming course in Haskell, Scheme or Eiffel will of course be creating such structures without wondering how they are built or reclaimed.

[2] cited in [Jones and Lins 1996; Zorn 1991] etc.

[3] Actually the original LISP system in 1958 included an *eralis* function for reclaiming memory, but this was quickly replaced with garbage collection.

badly managed heap can become fragmented, and unable to fulfil allocation requests even though sufficient free memory is available.

2.1.1 Allocation

Memory allocation is an extensive field, and the standard reference is the survey paper by Wilson, Johnstone, Neely, and Boles [1995]. The key issues for memory allocation are speed of allocation and de-allocation, and avoidance of fragmentation.

In an ideal system with unbounded memory resources, allocation could be done by incrementing a single pointer that indicates the highest address currently in use. This idea is referred to as *monotonic* or *bump pointer* allocation, and is of practical use in several situations. The most familiar is the stack pointer used to allocate local variables on procedure calls in traditional languages. In this case, all variables allocated on the stack are deallocated when the procedure returns to its caller, and certain copying garbage collection algorithms also make this a practical approach.

In systems (such as Lisp) where all memory cells are the same size, a bitmap can be used to indicate which cell of memory is free, and allocation involves searching the bitmap for the first free bit, then returning the address of the corresponding block.

The most widely used allocation strategy (at least in manually managed heaps) is the free-list allocator, which (broadly speaking) chains together free blocks of memory into a list. The design space for free-list allocators is very large, and involves complex tradeoffs between space and time efficiency.

One end of the design spectrum for free-list allocators is an *exact first fit* algorithm. Initially the heap is in a single large contiguous block, and allocation requests are satisfied by returning an initial segment of the unused memory. As blocks are returned to the memory manager, they are chained on a list, in ascending order of size. When a new request is processed, the free list is scanned, and the first block that is the same or greater size than the request is returned. If the block found is larger than the request, the block is split, and the unused portion is added to the free list. This algorithm suffers badly from external fragmentation, and *coalescing* of small unused chunks into larger fragments is needed to make this practical.

The other end of the spectrum is a *seregated fits* algorithm, that rounds up memory requests to a series of thresholds, for example powers of two. This type of algorithm suffers from extreme internal fragmentation, wasting in the worst case 50% of available memory. This approach is typified by the *Kingsley* allocator, a power of 2 segregated fits allocator, which is the default implementation of `malloc` in BSD Unix 4.2.

A *segregated free list* allocator maintains separate free lists for several different size classes. Many implementation variants are possible, but in the JMTk implementation, when a free list for a given size class is empty, a (4K) page of memory[4]. is allocated, and split into blocks of the free list size. One advantage of this scheme is that a per-page bitmap rather than a per-object header is sufficient to track objects in the heap.

[4]Not necessarily the operating system's virtual memory page size

The best known free-list allocators, such as the Lea allocator used by GNU libc, use a combination of strategies for different size objects. For small objects, which are most frequent, Lea uses segregated free-lists at 8-byte increments. For medium sized objects (64–128K bytes), a single free-list with approximate best-fit is used, while for objects larger than 128K, Lea uses the underlying operating system's virtual memory allocation functions.

2.1.2 Garbage collection

The purpose of garbage collection is to reclaim memory that is no longer needed by the program, and to recycle it for later use. Garbage collection algorithms can be classified on several attributes:

- How they identify live data, either by tracing (following pointers from a fixed *root set*) or using reference counts.
- Whether they run concurrently with the program, or as 'stop the world' collectors that interrupt program execution to perform collection.
- Whether they move objects in the heap (copying collectors), or not.
- Whether they require accurate type information from the compiler (accurate collectors) or can run with limited or no information (conservative and mostly-copying collectors).
- Whether they collect the whole heap, part of the heap (as in generational collectors) or spread the work evenly throughout program execution time (incremental collectors).

The time complexity of garbage collection algorithms is almost always linear in some aspect of program behaviour or heap size, and efficiency is gained by decreasing the constant factors in the complexity equation. Some collectors require intervention when values are stored or read from pointer fields, and have overhead proportional to the *poiner mutation rate*.

Garbage collection algorithms are covered in more detail below.

2.1.3 Terminology

The area of memory used to allocate dynamic data structures is known as the *heap*, and blocks of memory allocated on the heap are referred to as *objects*, whether the language is object oriented or not. The process of reclaiming unused memory is known as *Garbage collection*, a term coined in the original 1960 paper on Lisp [McCarthy 1960]. Following Dijkstra, Lamport, Martin, Scholten, and Steffens [1978], when looking at a program from the point of view of the garbage collector, the term *mutator* is used, being the part of the system that mutates the heap. Most garbage collection algorithms interrupt the execution of the mutator for varying amounts of time, and the term *pause time* is used to refer to the times when the garbage collector interrupts the main program.

2.2 Classical garbage collection algorithms

Most garbage collection algorithms in use today derive from one of three algorithms discovered in the 1960s.

2.2.1 Mark sweep

The first garbage collection algorithm was created as part of McCarthy's LISP system [McCarthy 1960], and is today known as mark-sweep collection. A single bit flag is associated with each object, initially set to 0. At the start of a garbage collection pass, the collector starts from a set of *root pointers*, and *traces* the heap. The root pointers are static program variables, stack pointers and registers. For each object traced, the system first sets the mark bit to 1, then iterates through all the pointer fields in the object, tracing the object pointed to. If a pointer refers to an already marked object, it is skipped and the next pointer is examined. This effectively forms the transitive closure of the root pointer set, identifying all objects which are reachable from variables in the program. All other memory locations are unused and may be reclaimed.

Once this *mark phase* is complete, the collector scans the heap sequentially, placing every object whose mark bit is 0 on a list of free memory cells, and setting the mark bit of live objects to 0 in preparation for the next collection phase. When this *sweep phase* is complete, control returns to the program and execution resumes.

Mark-sweep collection is relatively simple to implement, and has a low space overhead, requiring only a single bit per object. Implementations vary as to whether the mark bit is kept in the object header, or separately in a side array. Mark-sweep is relatively inefficient, taking time proportional to the size of the whole heap regardless of the amount of live data[5]. It also suffers from a large pause time, although there are concurrent mark sweep algorithms that reduce this. Mark-sweep collection requires a free-list memory organisation, and therefore incurs a higher allocation cost than garbage collection schemes that use monotonic allocation.

2.2.2 Reference counting

Reference counting was the second garbage collection algorithm published, also in 1960 for the LISP system [Collins 1960]. Each object is extended with a field that counts the number of pointers that refer to it. When the count drops to zero, the object is no longer required, and can be reclaimed. When an object is freed, objects that it refers to also have their reference counts decremented. Implementing reference counting requires a *write barrier* to be inserted before code that manipulates pointers, so that the counts can be adjusted and freed memory reclaimed.

The main advantage of reference counting is that there is no delay between memory becoming free, and it being available for re-use. Applications using reference counting can operate efficiently in heaps only slightly larger than the peak size that

[5] Although lazy sweeping techniques [Jones and Lins 1996] reduce this problem.

the application requires. The pause times of reference counting collectors are also proportional to the size of data structures that are freed, which are typically much smaller than the whole heap, and lazy reclamation techniques have been developed that only ever require single objects to be freed at any given time.

One disadvantage of reference counting is that pointer mutations are frequent and the write barrier is expensive. Thus straightforward reference counting implementations are generally among the slowest of memory management schemes. The other significant disadvantage is that reference counting cannot in itself collect cyclic data structures, and alternative approaches such as 'trial deletion' or mark-sweep collection must be employed. Trial deletion involves identifying candidate objects for the roots of cycles, tentatively setting their reference count to zero, and seeing whether this would have caused the object to be collected had its reference count not been decremented. The cost of trial deletion can be a significant effect on program run time [Quinane 2003], while using alternative techniques obviates the need for reference counting at all.

Deferred reference counting [Deutsch and Bobrow 1976] can significantly reduce the overhead of standard reference counting. Collection is performed in distinct passes (as in other collection algorithms), and mutations to certain frequently changed roots (such as registers, the stack etc.) are only counted during a collection cycle. Particularly when combined with coalescing [Levanoni and Petrank 2001; Warrington 2003], deferred reference counting once again becomes a viable technique.

2.2.3 Copying

Copying collectors were proposed in 1969 by Hansen [1969] and Fenichel and Yochelson [1969], and first became practical with Cheney's algorithm [Cheney 1970]. Copying collection combines the advantage of low mutator overhead with a collector whose cost is proportional to the amount of live data in the heap. Since most heap objects are short lived, this can be considerably more efficient than mark-sweep collectors whose performance is proportional to the total size of the heap.

The original copying collector is known today as a semi-space collector, because it divides the available heap space into two equal sized regions. Using a bump-pointer allocator, objects are created in one half of memory, known as *to-space*. When all memory in to-space is consumed, the unused half of memory is renamed *to-space*, and the old to-space becomes *from-space*. Starting with the set of root pointers, from-space is traced as in a mark-sweep collector, but instead of simply being marked, each live object is copied to to-space. A bit is set in the old object (to mark it as having been copied), and a forwarding pointer is written to it to indicate where it is in to-space.

As pointers are traced from live objects, either an object is still in from-space, in which case it is copied and the pointer is updated to point to its new location, or it has already been copied in which case the forwarding pointer is used to just update the pointer being traced.

Cheney's algorithm uses the set of already-copied objects as a work queue to eliminate the need for additional data structures, thus making copying a practical technique by bounding the overhead required.

The attraction of copying collectors is that in typical programs, typically less than 10% of the heap is live. Thus copying, which has a cost proportional to the size of the live objects in the heap, is more efficient than an algorithm like mark-sweep. Copying collectors also permit the use of a bump-pointer allocation scheme, also increasing program efficiency. Another advantage of copying collectors is that they compact the heap, improving locality.

The disadvantage of copying collectors is that time is often wasted copying long-lived data structures from one semi-space to the other and back.

Later copying collectors include several compacting algorithms which use copying but within the same space, reducing space overhead at the cost of efficiency.

2.3 Modern algorithms

Modern algorithms use elements of the three classical algorithms, but are based on various observations about the statistical distribution of lifetimes and/or operations on objects in a typical heap.

2.3.1 Generational

Two hypotheses were posed in the early 1980s regarding the lifetime of objects in a typical program. The *weak generational hypothesis* [Ungar 1984] proposes that 'most objects die young', and is a generalisation of the *strong generational hypothesis*, that object lifetime is proportional to age, ie that object lifetimes exhibit an inverse exponential distribution. Observation shows that in general the weak generational hypothesis holds for many workloads, whereas the strong generational hypothesis does not.

Generational collectors [Lieberman and Hewitt 1983] divide the heap into regions for objects of different ages, and perform more frequent collections on more recently allocated objects, and less frequent collections on the oldest objects. The youngest generation is generally known as the *nursery*, and a collection that collects only the nursery is known as a minor collection. During a minor collection, pointers from older generations into the nursery are used in addition to the standard root set when tracing the set of live objects. Although it is possible (but expensive) to calculate this set at collection time, this set is generally maintained using a *generational write barrier* that takes note of pointers from older generations to younger generations, keeping them in the *remembered set* for use during minor collections.

Generational collectors are very effective. The majority of collectors in practical systems implemented today are a variety of generational collector.

2.3.2 Older-first

A nursery collection in a generational collector will tend to copy objects that have lived for a short time, and which can be expected to live longer. Unfortunately, it will also copy the most recently allocated objects, many of which will only live for a very short time. The aim of the *older first* collectors is to give objects time to die, and so to only preserve (through copying or otherwise) those objects that will live the longest.

Clinger and Hansen [1997] were first to suggest that a generational collector could be improved on by collecting an older portion of the heap. This idea has been developed in [Stefanović et al. 1998; Stefanovic et al. 1999], and has led to collectors such as the collector for Scheme described in [Clinger and Hansen 2002].

2.3.3 Beltway

Beltway [Blackburn et al. 2002] generalises over many other collection algorithms, including semi-space, simple generational and older-first collection.

2.3.4 Hybrids

The original generational collectors used pure copying techniques; surviving objects from younger generations are copied to the next older generation, and the oldest generation treated as a semi-space. The observation that object dynamics differ between different generations of objects leads to the conclusion that different algorithms can be used to manage them. Nursery objects, for example, are rapidly allocated and mutated, and quickly die, making them suitable for copying collection and bump-pointer allocation. Mature objects have lower mutation rates, and are allocated and deallocated less frequently, making them suited to other collection techniques. Several algorithms have been explored, incuding reference counting and mark-sweep collection, coupled with a copying nursery. These algorithms are the fastest and most responsive known today.

Attanasio, Bacon, Cocchi, and Smith [2001] give positive performance measurements for a hybrid copying/mark-sweep collector, where live objects in the nursery are copied into a mark-sweep older generation. Copying is well suited to the nursery generation, where a large proportion of objects die before being copied, and mark-sweep avoids copying long lived objects multiple times.

Blackburn and McKinley [2003] study a hybrid copying/reference counting collector, obtaining similar throughput to Attansio et al's collector, but with much improved pause times.

2.4 Conservative collection

All algorithms discussed above are *accurate*, in that they use information provided by the compiler to accurately determine which memory locations are pointers. Languages such as C and C++, which allow arbitrary type casts and pointer arithmetic, are fundamentally *GC-unfriendly*. Garbage collection of such languages is still possible, however, with *conservative* algorithms.

The original—and still most widely used—conservative collector is the Boehm-Demers-Weiser collector [Boehm and Weiser 1988]. This collector uses heuristics to identify which words in the stack, global variables and other areas (machine dependant) could possibly be heap pointers. These values are used as roots and the heap is traced, regarding all words in allocated objects in the heap as possible pointers. Because the collector is never sure which values are actually pointers and which are non-pointers, the collector must never move objects, and the Boehm collector uses a mark-sweep algorithm.

In the design space between the Boehm collector and fully accurate collectors are *mostly copying* collectors, originally introduced by Bartlett [1988]. These collectors assume no type knowledge of the stack, registers or global variables, but rely on the compiler for information about the layout of heap objects. The technique is suited to compilers that use C as an intermediate language, and can thus control the contents of the heap, while the contents of the stack etc is under the control of the underlying C compiler. Mostly copying collectors take advantage of this by using copying techniques for objects in the heap that are not pointed to by the (ambiguous) root set.

Many modern compilers for languages other than C and C++ use the Boehm collector despite the fact that they could use accurate techniques. Little is known about the relative performance of the Boehm collector and accurate techniques.

2.5 Performance

Despite the copious literature, little is known about the relative performance of garbage collection algorithms. Jones and Lins [1996] spend 3 pages discussing the difficulty of comparing garbage collection algorithms, both from the theoretical standpoint that different systems have different criteria, and the practical difficulty of comparing different systems implemented in different languages in subtly different ways. Wilson [1994] shows that most studies of the performance of garbage collectors to that date had serious limitations, and concludes that more studies of garbage collection in high performance systems were called for.

Most papers describing garbage collectors either implement them in the context of a new language or compiler, or are implementing a collector to replace the existing one in a given system. In the first case, there is very little basis for comparison, and the secodn is scarcely better. The effort involved in constructing a memory management subsystem from scratch make it difficult to build multiple algorithms for comparison purposes.

Some examples from the literature will server to illustrate the problem. Appel [1989] describes a variable size nursery generational copying collector, one of the fastest garbage collectors known today, and simply estimates performance by looking at instruction counts and frequency of updates. Sansom [1991] describes a dual-mode collector which combines copying and sliding compaction in a generational collector, and benchmarks it in the Glasgow Haskell Compiler against the semi-space collector it replaces. This collector does not seem to have been implemented anywhere else. Clinger and Hansen [2002] compare 3 collectors (2 generational copying collectors and a renewal older first collector) in the Larceny Scheme compiler. It is impossible to tell whether the performance results seen are simply artifacts of the implementation techniques, or of the programming language or benchmarks chosen.

Blackburn, Cheng, and McKinley [2004a] compare six collectors in the same Java virtual machine using the Spec JVM 98 benchmark suite. This is the most comprehensive study of garbage collection algorithms, although open questions still remain as to whether their results would hold in another language.

Despite the large body of research, many performance questions in the field are still open. A portable toolkit for garbage collector development will help to fill this gap.

2.6 Toolkits

This project is based on the JMTk memory management toolkit [Blackburn et al. 2004b], a memory management toolkit described in detail in Chapter 3.

The majority of garbage collection and memory management systems are written as monolithic programs, generally in C, as common wisdom holds that such low-level code should be written in a language such as C, designed for the purpose. There have been a small number of garbage collection toolkits written, as identified in [Blackburn et al. 2004b].

The UMass Language Independent GC Toolkit [Hudson et al. 1991; Stefanović 1993] was the first system to provide a reusable infrastructure for garbage collector development. GCTk [Blackburn et al. 2002] was the first garbage collection toolkit to be written in Java, providing some of the functionality of JMTk, but without free-list allocation or JMTk's composability.

2.7 Conclusion

Automatic memory management is an active area of research, both in terms of new algorithms being discovered, and old algorithms re-evaluated. In particular, it is only recently that many garbage collection techniques are being tested in the same environment and with the same workload. Garbage collection infrastructure, in the form of toolkits for garbage collector development, is a valuable research tool, and language independent toolkits which allow even broader comparisons to be made will be very useful indeed.

The next chapter describes JMTk in detail, and outlines the problems that are faced in making it portable.

Chapter 3

Porting JMTk

JMTk is a memory management toolkit that currently runs on a single platform, the Jikes RVM Java Virtual Machine. Many of the gaps in modern memory management research are due to the complexity of garbage collectors. Implementing a particular algorithm for a given language runtime is a significant project, and a researcher who sets out to compare several algorithms within the same environment is faced with an almost insurmountable task. A portable, efficient toolkit for building memory managers would be a significant enabler for research, and that is the aim of this research project.

This chapter introduces JMTk and its current host environment, Jikes RVM; briefly describes the target platforms for the porting project; and outlines the approach that was taken in making JMTk portable. The previous chapter has given the broader background of memory management and garbage collection; the role of this chapter is to introduce the specific elements of the project and to set the scene for the main body of the thesis.

3.1 JMTk

JMTk[1] (Java Memory Management Toolkit) [Blackburn et al. 2004b; Blackburn et al. 2004a] was designed and implemented by Steve Blackburn, Perry Cheng and Kathryn McKinley for the Jikes RVM Java virtual machine.

JMTk provides reusable components at multiple layers of abstraction, that can be composed into memory management *plans* for integration into a programming language runtime. Existing plans provided by JMTk are:

semiSpace a classic semi-space copying collector

markSweep a classic mark-sweep collector

refCount deferred reference counting

[1] Soon to become MMTk

copyMS a simple generational collector, with a bump-pointer nursery and a mark-sweep mature space, but without a write barrier and remsets etc. All collections involve tracing the entire heap.

genCopy a classic copying generational collector, complete with a write barrier and remsets.

genMS a generational collector with a bump-pointer nursery and a mark-sweep mature space, complete with a write barrier and remsets.

genRC a generational collector with a bump-pointer nursery and a reference counted mature space. Described by Blackburn and McKinley [2003].

JMTk makes considerable use of Java's object oriented programming facilities to provide its functionality in a simple and easily extended way so as to fulfil its objectives as a platform for research.

JMTk provides re-usable components at several levels. At the lowest level of functionality are utility classes, providing the basic building blocks of higher level functions, themselves factored into as many reusable sub-components as is practical. For example, double-ended queues are provided abstractly, unsynchronized (for local per-processor use), synchronized (for global use), and for three datatype combinations. Other utility classes include bump-pointer allocation, free-list allocation, remembered sets etc.

At the next level of complexity, JMTk provides management policies for virtual memory regions, such as copy spaces, mark-sweep spaces, reference-counted spaces. At the highest level, JMTk provides memory management *plans*, which are fully-formed memory management algorithm implementations.

JMTk has been constructed with portability in mind. To a large extent, when porting commenced, the Jikes RVM specific code was isolated in a separate interface package, called `vmInterface`. This package contains the virtual-machine specific 'glue' code that connects the VM-neutral JMTk code with the host virtual machine.

3.2 Jikes RVM

Jikes RVM (formerly Jalapeño) [Alpern et al. 2000; Alpern et al. 1999] is a research Java virtual machine developed at IBM's Thomas Watson research labs. One distinctive feature of Jikes RVM is that it is written almost entirely in Java, rather than C or C++; the language of choice of most Java virtual machine implementations.

The safety features of Java are a barrier to the implementation of low-level code, and Jikes RVM uses special Java idioms, described by Alpern, Attanasio, Barton, Cocchi, Hummel, Lieber, Ngo, Mergen, Shepherd, and Smith [1999] to work around this limitation rather than use the standard approach of creating native methods to do the low-level work [Ritchie 1997]. The mechanisms are:

Magic classes Methods and objects of these classes are inlined by the compiler with nonstandard representations. One example is the `VM_Address` class, which rep-

```
private final void writeBarrier(VM_Address src, VM_Address tgt)
  throws VM_PragmaInline {
  if (src.LT(NURSERY_START) && tgt.GE(NURSERY_START)) {
    remset.insert(src);
  }
  VM_Magic.setMemoryAddress(src, tgt);
}
```

Figure 3.1: A code fragment from JMTk illustrating the Jikes RVM idiom.

resents an address. Instances are replaced in memory by a single machine address, and instance methods (add, subtract etc) are replaced by the equivalent machine instruction.

Magic Methods There is also a `VM_Magic` class, with static methods that represent specific machine instructions, such as memory barriers or atomic test-and-set sequences, direct memory access, unchecked type cast etc.

Pragma exceptions Certain exceptions can appear in the `throws` clause of a method, and these can influence the way the compiler generates bytecode for the method. These exceptions are all of the form `VM_Pragma<x>`, where `<x>` is:

Inline	The method is guaranteed to be inlined
NoInline	The method is guaranteed not to be inlined
Uninterruptible	The method is atomic within the java threads model; the compiler does not generate yield points or stack checks
Interruptible	In a class that implements VM_Uninterruptible, this pragma relaxes the synchronization requirement
LogicallyUninterruptible	Uninterruptible, but disabling some compiler checks for uninterruptible methods
NoOptCompile	Disables the optimizing compiler for the method

The idiom approach to implementing low-level features in Java is shared with the OVM Java virtual machine, and the concept is formalised and expanded on by Flack, Hosking, and Vitek [2003]. An example of Java code using the Jikes RVM idiom is given in Figure 3.1. The illustrated code is the default generational write barrier, and demonstrates the use of: the `VM_Address` class (the method calls to the `LT` and `GE` methods compile to a single instruction each); a pragma exception to force this method inline; and a call to `VM_Magic`, in this case a direct memory store.

The magic types provided by Jikes RVM are:

`VM_Address` An address type, with methods for address arithmetic, comparison and casts to other size-equivalent types.

`VM_Offset` An offset in memory, ie a 32- or 64-bit signed magnitude depending on the address size of the target architecture.

`VM_Extent` An unsigned magnitude, representing the size of something in memory. 32- or 64-bits as required.

`VM_Word` A word of memory, of the natural size for the architecture. Bitwise operations are defined on `VM_Word`s.

`VM_AddressArray` An array of `VM_Address`es.

The porting process must allow for these features of the JMTk code, either translating them into a functional equivalent, or removing the construct if its functionality is unsupported or not required.

3.3 Target platforms

The port of JMTk was done with two target environments in mind.

3.3.1 nhc98

Nhc98 ('Nearly a Haskell Compiler') is (despite the name) a fully fledged Haskell 98 compiler. Currently nhc98 is maintained by the Functional Programming group at the University of York, and the main focus of development is memory efficiency in functional programming.

The nhc98 runtime is based on the G-machine, a virtual machine for graph reduction [Peyton Jones 1987], designed by Augustsson [1984] and Johnsson [1984] to implement Lazy ML (LML). The nhc98 runtime has the advantage of simplicity, although its execution time performance is significantly slower than that of other Haskell compilers, and memory management time makes up only a small proportion of the total runtime of programs.

The nhc98 runtime is written in C, and built with gcc. Nhc98 only supports single-threaded execution.

3.3.2 Ghc

The Glasgow Haskell Compiler (ghc) represents the state of the art in compilation of Haskell, and indeed of lazy functional languages. Ghc compiles to machine code using gcc as a portable intermediate code.

Ghc implements a superset of Haskell 98, and from the point of view of the memory management subsystem, the two most significant features are stable pointers (object references that can be passed to functions in different languages, hashed etc), and weak references. JMTk provides mechanisms that should be able to implement both of these features, but they were not explicitly considered in this project.

Ghc is discussed in detail in Chapter 6.

3.4 The approach

The approach taken to porting JMTk is to keep a common Java code base for all target platforms. Changes required on a platform specific basis will be confined to a separate interface package, and the interface will be kept as small as possible. In order to port the nonstandard features of Java in the Jikes RVM environment, a source code transformation (discussed in Chapter 5) was used, and the transformed code was compiled using the gcj compiler.

3.4.1 gcj—the GNU Compiler for Java

Gcj is an ahead-of-time compiler for Java in the gcc (GNU compiler collection) family of compilers. Gcj provides a front-end for the gcc code generator and generates moderately good code on a large range of platforms.

Gcj provides the standard JNI interface for calling code written in C, but also provides a new interface, CNI, for interacting with C++. CNI is claimed to be much more efficient than JNI, as the compiler uses the same object layout and calling conventions for C++ and Java.

3.5 What lies ahead

This part of the thesis has introduced the project, given some background material on memory management, and described the target of the project (JMTk) and the challenges faced in making JMTk portable. The second part of the thesis will describe how the difficulties were overcome and how the project was implemented, while the third and final part will presents results and conclusions derived from the project.

Part II

Body

Chapter 4

Virtual machine interface

Porting JMTk to another runtime environment requires identifying and isolating the places where it depends on Jikes RVM functionality, and building a new interface layer that will provide the required features in the new environment. This chapter describes the work done to separate virtual machine dependencies into an interface layer, and how the new interface layer was built. This chapter covers the refactoring of JMTk, the programming language architecture of the interface layer, and the design and construction of a testbed environment for the portable version of JMTk.

This chapter covers moderately low-level detail of the code produced for the interface, as this is where the issues of interest lie.

4.1 Architecture of a portable JMTk

The ideal portable memory manager could be distributed as a `.jar` file (this being a Java system), and simply included at runtime, and would have a clean, simple API. The Boehm conservative collector certainly approaches this ideal, as it is distributed as a `tar` archive, builds with the traditional `configure; make install` sequence, and requires only that memory be allocated through calls to `GC_MALLOC`.

Three issues prevent JMTk from achieving such simplicity. Firstly, the Java language does not provide the native facility for handling type-unsafe address datatypes, requiring different approaches for different systems. This issue is the main topic of the transformation discussed in Chapter 5. Secondly, accurate collection requires much closer cooperation between the language runtime/compiler and the memory management subsystem. Issues such as how data types are laid out in memory (the *object model*), how to identify pointer words in objects when the heap is traced (*object scanning*) and where to find root pointers are very language environment specific, and providing a virtual machine independent interface is difficult. Thirdly, JMTk makes no assumptions about the services provided by the enclosing runtime, and its interface exposes all of JMTk's requirements, even those that could be expected to be provided by the underlying operating system. This provides for greater portability than simply assuming a Unix API.

The JMTk approach is to provide an interface layer that sits between the runtime system and JMTk, translating JMTk's requirements of the host runtime into the services the runtime provides, and implementing the runtime's requirements in terms of JMTk's public interface. In the case of JikesRVM and other Java virtual machines, the interface layer is a Java package, `vmInterface`, while in this project the layer is a mixture of Java, C++ and C code, designed to interface with runtime systems implemented in C. Section 4.2 describes how the interface separation was enforced in the original JMTk code, Section 4.3 describes the issues raised by the multi language environment of the ported system, and Section 4.4 looks at the interface built for the JMTk test environment.

4.2 Refactoring JMTk

When work began, most of the platform dependencies were isolated into a separate Java package, the `vmInterface` package. Many dependencies remained, however, in the core JMTk code.

The first task in performing the port was to isolate the remaining JikesRVM dependencies in the main JMTk package into the `vmInterface` package, and determine which features would be subject to source transformation. The methodology for this is straightforward, if time consuming.

Firstly, dependencies were identified using a script that stripped out `import` clauses in the source, and formatted into a dependency matrix, with rows for JMTk classes, and columns for the JikesRVM classes they depended on. Using this matrix as a guide, a list of methods imported from each JikesRVM class was compiled.

One of the principal portability challenges is the 'magic' classes that JMTk uses to manage raw addresses, bit-patterns etc. The Jikes RVM idiom treats these entities like standard objects, but with the guarantee that their concrete implementation is as a raw pointer. The original design idea had been that only these machine-word classes would be dealt with by source transformation, and this was quickly vindicated by examination of the full set of dependencies. The remaining dependencies were dealt with by a variety of techniques:

- Replacing the call by an indirect call through the `VM_Interface` class[1]. Jikes RVM's inlining capabilities ensure that this is never a performance issue.

- Moving a whole class into the `vmInterface` package. Classes such as 'Lock' (which provides synchronization primitives) are almost entirely system-dependent.

- Abstracting the mechanism so as to remove or isolate the system dependency. The `Log` class discussed below is an example of this.

Some examples merit individual discussion.

[1]This class contains the majority of the interface methods.

4.2.1 Logging output

Jikes RVM provides a highly overloaded method, `VM.sysWrite`, which writes a message to the system console for debugging or error messages. The large number of overloads is because it supports writing sequences of different datatypes, for example, a string, and integer, a string and a float. The `sysWrite` method provides atomicity: the output stream is locked while all the arguments to a single `sysWrite` call are written. Since string concatenation (the standard Java approach to the same problem) requires memory allocation and would not be possible in most parts of JMTk, this call is essential and widely used.

Requiring an implementor to provide 60 or more variants of a single method, most of which are boilerplate constructions from the simple base cases seems a little unreasonable. The solution to this problem was to develop a `Log` class with a single-argument overloaded `write` method, that buffered output flushing it to the output stream on either buffer overflow, an explicit request or a `writeln` call. The resulting style of programming would be familiar to Modula-2 programmers worldwide. The problem was identified by myself, and the `Log` class was implemented by Andrew Gray from a design by Steve Blackburn.

Implementing the `Log` class duplicates code provided in JikesRVM, but succeeds in reducing the size of the virtual machine interface (by over 60 methods) without exposing the programmer to locking protocols and further potential sources of program error. The resulting `Log` class requires a single `VM_Interface` method to write an array of char atomically to the output stream.

4.2.2 Interface separation

When work began, the majority of the functionality of the `vmInterface` package was provided by a single class, `VM_Interface` which provided all interface methods required by JMTk and Jikes RVM. One early step in the porting process was to split `VM_Interface` into 2 classes, `VM_Interface`, the JMTk-to-Virtual-Machine interface, and `MM_Interface`, the Virtual-Machine-to-JMTk interface. Closer examination of the architecture created by this separation leads to the specification of 4 separate interfaces, as shown in Figure 4.1

The `MM_Interface` class defines the facilities provided to the virtual machine (interface 'b' in Figure 4.1). This interface is tailored to the requirements of the specific virtual machine, and are defined in terms of the public interface (in Java terminology) of the JMTk package (interface 'a'). The `VM_Interface` class defines the facilities required by JMTk (interface 'c'), and is defined in terms of whatever facilities the virtual machine environment exposes for JMTk to use (interface 'd').

The real situation is more complex than this: there are additional classes in the vmInterface package, but these either broadly fit the description of either `VM_-` or `MM_-Interface`, or are purely internal to the interface package, and occupy ground between the `VM` and `MM` interfaces. Examples of this are:

`ScanObject` This class provides JMTk with the capability to scan the pointer fields

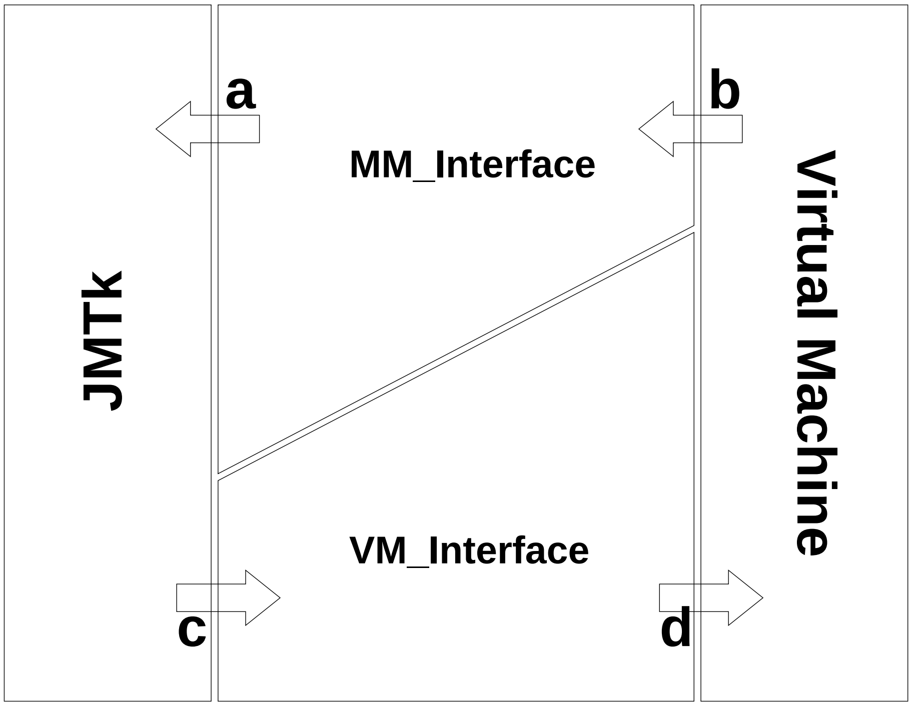


Figure 4.1: JMTk interface structure

of objects and call back to a plan-specific method for each one (via the `Enumerate` class). Conceptually this class fits the description of `VM_Interface`.

`ScanStatics` This class (in the Jikes RVM version of vmInterface) enumerates pointers in static objects analogously with ScanObject for dynamic objects. The key difference is that this is encapsulated in the `vmInterface` package, and called via the `computeAllRoots` method of `VM_Interface`. This can be regarded as a sub-component of `VM_Interface` that only interacts with interface 'd'.

`Util` Provides common utility methods for other `vmInterface` classes, but has no public methods.

Separation of the interfaces is an essential step in porting of JMTk, as methods in the `MM_Interface` are virtual machine environment specific, and need not be implemented by the new interface package.

4.2.3 Object Model

The term *object model* refers to the way objects (both in the specific sense of object-oriented languages, and the more general sense of heap objects in other languages) are laid out in memory, how their type is determined and how they are manipulated by the runtime and memory management systems. Creating an abstract interface to the object models of multiple virtual machines is a significant challenge.

From the point of view of the memory manager, an object has a size, a set of pointer fields, and some metadata fields where the garbage collector can store the information it needs to manage the heap. Concrete implementations of object models vary considerably, and possible designs include: storing all metadata in the object itself, in a header structure; storing a single pointer (or index into a table) in the object to a structure that describes all objects of a given type; and storing objects of different types in different virtual memory locations, so that the object pointer itself defines the structure of the object. One common technique for storing small metadata fields is called *bit-stealing*, and takes advantage of the alignment of data structures to 2, 4 or 8-byte boundaries. Where the object contains a pointer to an aligned structure, the least significant bits of the pointer are always zero. Therefore, the run-time system can store a small metadata field in the low-order bits, provided that the runtime system masks forces the bits to zero before using the pointer value. Metadata can be per-class (per-type), or per-object, and most garbage collection algorithms require at least one bit of per-object metadata. For example, an object model using a single pointer to a per-class structure can support a narrow bit field (either as a mark bit or a copy-state bit) via bit-stealing, and can overwrite this field with a forwarding pointer, as object layout is no longer important once an object has been copied.

Jikes RVM supports a number of object models, and allows different parts of the virtual machine, including the memory manager, to provide elements of an object header, whose layout is determined automatically at build time. The interface provided to JMTk allows access to an 'available bits word' containing a small bit-field where it can store metadata and with the remainder of the word available for use as a forwarding pointer. Other metadata fields are defined by the memory management portion of the object header. This approach clearly does not translate to a multi-language environment where the object header could be laid out in a C `struct` definition.

In terms of pointer layouts, Jikes RVM supports two types of object: scalars and arrays. Scanning an array (of a reference type) simply requires knowing how many elements in the array, and the offset to the first element of the array, and this information is held in the header of the object. Pointers in scalars (ordinary class objects) are laid out in the order of their declaration in the class, and when a class is loaded, Jikes RVM builds a structure describing the class for the garbage collector. This consists of an array of offsets into the object, pointed to by the 'type information block' that is built to describe each class.

The target architecture, ghc, contains objects with layouts not found in Jikes RVM. Most objects in ghc's runtime are *closures*, consisting of a fixed header, a number of pointer fields, and a number of non-pointer fields. In principle, most objects can be described by two integers: the number of pointer fields, and the offset from the start of the object. Ghc also supports more complex objects, such as partial applications (known as *PAPs*), which contain chunks of the evaluation stack, where pointers are identified by bitmaps contained in the object.

Work on abstracting the object model is still progressing, but two steps have been taken so far. A new mechanism for scanning objects has been developed, discussed in detail in section 4.4.2. The portable version of JMTk has been built using a new, consolidated interface to the object model. This interface abstracts the metadata fields into a numbered set of fields designed for different purposes, not all of which will be defined in any given concrete implementation. This interface is still in development, and has not yet been adopted by the main JMTk code.

4.3 Language architecture

The GNU Java compiler, gcj, provides 2 cross-language interfaces: JNI, the standard Java Native Interface, and CNI, the Cygnus Native Interface. JNI is a more cumbersome facility (for instance methods implemented in native code must be compiled to a shared object module), and the gcj documentation claims it is much less efficient than CNI, as well as being less flexible and powerful.

The CNI interface is based on the observation that apart from a handful of key differences, Java is essentially a subset of C++, and in the gcj compiler the same calling conventions are used. If a static method in Java is declared `native`, the linker will match this with a C++ function in the appropriate namespace. For example, the Java code fragment

```
package pkg;
class cls {
  static native void m1(int i);
}
```

will (at link time) match the C++ code

```
namespace pkg {
  namespace cls {
    void m1(jint i) {
      ...
    } } }
```

Java classes declared as `extern "Java"` can be accessed by C++ code as though they were C++ classes.

The one disadvantage of this embedding is that when C code needs to call Java code (eg in the performance critical allocation or write barrier code), a C++ 'impedance matching' layer is required to translate between the flat namespace that C inhabits and the embedding of Java in C++.

4.3.1 Language boundaries

Because of the performance penalty of crossing the langauge boundary, the decision as to which components of the interface are in what language is significant. The decision made when planning the ghc port, and carried through into the testbed architecture was that all code pertaining to the structure of roots and objects would be kept in C (or C++ where a header file was all that required accessing). This approach would maximize the possibilities for code re-use between the current ghc garbage collector and JMTk. The alternative would have been to build a Java class that duplicated all the object structures that the runtime contains, and given the complexity of some objects (as discussed above) this would yield some very complex Java code. In keeping with the guiding philosophy of 'tune later', the performance issues surrounding the boundary crossing were left until later.

Some functions of `VM_Interface` can only be provided (under gcj) via a C/C++ program to operating system/glibc functions. In particular, the mmap, munmap, mprotect and memcpy functions are accessed via a native version of `VM_Interface`.

4.4 Testbed architecture

One key design decision for the porting process was that a standalone testbed environment would be built so that the ported JMTk could be tested independently from a given language runtime. The testbed was designed so that the ghc runtime interface would share much of the interface code for the testbed.

The testbed must provide a simple object model, with the support code for scanning objects and accessing metadata. It also provides a root set and scanning code, plus an interface for the testbed to call for allocating memory, requesting garbage collections etc.

In order to move beyond the simplest of tests (allocate an object; do a collection etc), a somewhat artificial programming style needs to be adopted, since we are programming in an environment where the language (C) provides no support whatsoever for garbage collection.

4.4.1 Testbed interface

A block diagram of the testbed is included as Figure 4.2, which for reasons of clarity only shows a subset of the modules that comprise the testbed system. The naming convention is that where a Java source file, `<class>.java` has a native component the native file name is `<class>Native.cc`.

The gcj interface comprises the following modules[2]:

`VM_Interface` The required `VM_Interface` class. The corresponding native module implements the low level memory protection and copying functions.

`ObjectModel` The required `ObjectModel` class. All functions are provided by the corresponding C++ module.

`ScanObject` The required `ScanObject` class. Described in detail in section 4.4.2 below.

`ScanRoots` Used internally by `VM_Interface` to enumerate root pointers. Described in section 4.4.2 below.

[2]Where not otherwise specified, modules are Java files and would have a `.java` suffix.

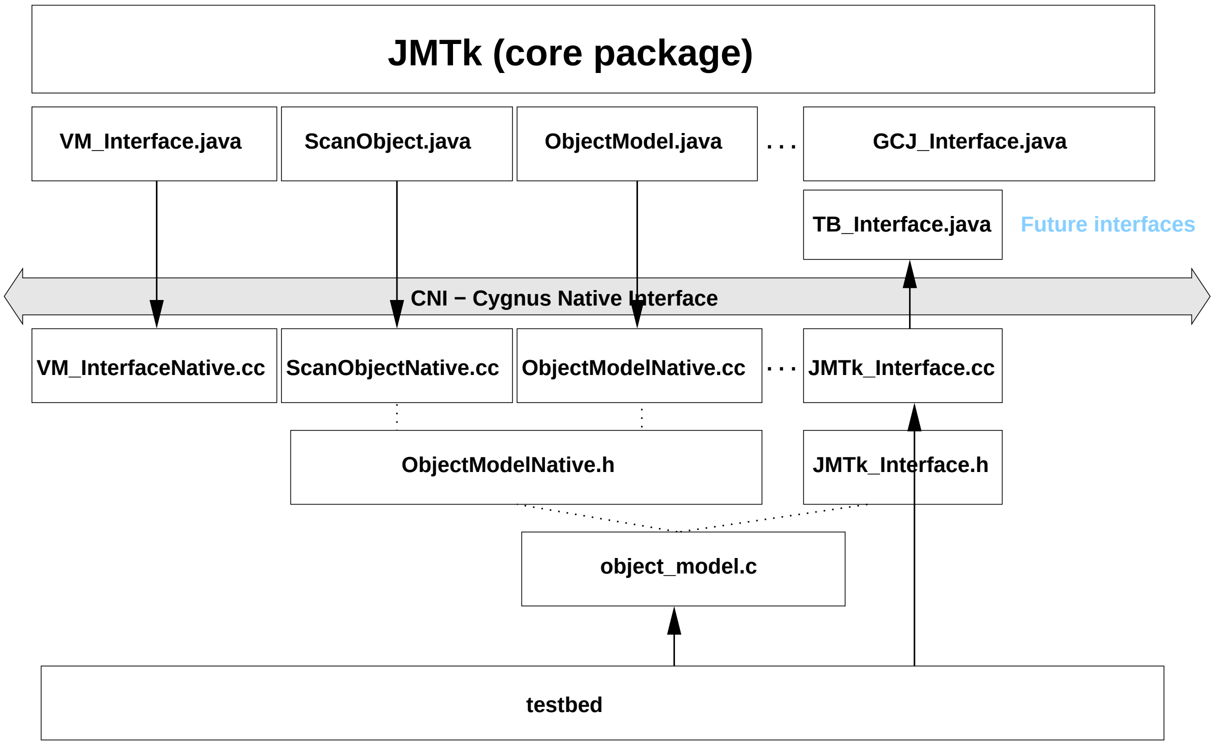


Figure 4.2: Testbed interface architecture

`GCJ_Interface` Provides low-level functionality to all gcj-based clients of JMTk.

4.4.2 Object scanning

When a tracing collector is traversing the heap, it visits each object, performs whatever operation is required for live objects in the heap, and then visits all of the objects pointed to by the current object. In JMTk, this is done by a method that pushes the addresses of all pointers in an object into a queue for later processing, and this operation is known as *scanning* the object.

In section 4.2.3, some of the object formats in Jikes RVM and ghc were described. The variety of possible object layouts means that it is clearly impractical for JMTk to know how to scan all objects in all possible virtual machines, but on the other hand making scanning of objects a virtual machine responsibility is also unpalatable.

The initial implementation of object and root scanning delegated the entire operation to the native code, which called back to a Java method for each pointer field. A more elegant and potentially higher performance approach has been implemented which should allow some of the current interface code to be taken back into JMTk proper. The approach takes advantage of the interoperability with C++ afforded by the CNI interface.

When JMTk wants to scan an object, it requests a 'helper' object (that inherits from the abstract `ScanHelper` class), and the helper identifies the pointer fields in the object. One distinguished subtype of `ScanHelper` is `ScanHelperDelegate`, which indicates that the virtual machine would like to take responsibility for scanning the object: an appropriate native method is called, which calls back to process each pointer. All other sub-classes of `ScanHelper` provide a `getElements()` method to return the number of pointer fields in the object, and a `getAddress` method which returns the address of the *n*th pointer in the object. Essentially a ScanHelper is an iterator for an object[3].

The intention is that at most one helper should be required per object type in the running program, and could be generated by the compiler or by the JMTk 'glue' initialization. The call to get a helper for a scan will identify the type of the object, and return the statically allocated helper object.

The various subtypes of `ScanHelper` encapsulate patterns of object layout:

`ScanHelperArray` scans an array-like object, ie a contiguous range of pointers at a given offset from the object header. Most conventional Haskell objects could be scanned by this kind of helper.

`ScanHelperOffsets` provides an array of offsets to the pointers in an object. Jikes RVM generates such arrays for scalar objects.

Other helper objects could be created, eg for stack frames described by bitmaps.

Scanning of roots is implemented using the same mechanisms, but using what amounts to an iterator to return base addresses and helpers for various root structures. The idea is that the roots are essentially a set of structures in patterns similar to heap objects. Where the structure of the root objects doesn't fit one of the available patterns, delegation is still available. Haskell stacks (known as *TSOs*: thread state objects) would be one good example where delegation might be appropriate, although a variety of bitmap helpers may suffice.

The value of this abstraction is that the helper objects give JMTk enough information that scanning of objects can take place entirely in Java code, once the initial call to fetch a helper has completed. This opens up the possibility of compiler optimizations that would not be possible if language boundaries have to be crossed for each pointer scanned.

4.4.3 Testbed proper

The role of the testbed is to exercise all functions of JMTk in an environment that is as simple as possible. Debugging garbage collectors and memory managers in particular is a notoriously difficult task. In Jikes RVM, where JMTk is compiled by a compiler that itself uses JMTk for memory allocation, heap corrupting bugs can be very difficult to track down. The testbed environment aims to simplify the testing process.

The testbed provides tests of varying degrees of complexity. The simplest tests simply allocate objects of varying sizes. Intermediate tests perform allocation and

[3] Implementation considerations prohibit actually implementing the Java iterator interface, which is inherently stateful.

garbage collection, and examine the state of the heap at each stage. Currently the most complex tests implement algorithms that manipulate large data structures, and which are being used as benchmarks for the performance of the ported system.

In order to fulfil its role as a test platform, the testbed allows extreme levels of verbosity, for example logging every read and write of object metadata fields.

The main program `main.c` of the testbed is a simple test harness that processes command line arguments and calls tests defined in separate modules. The command line syntax of the testbed is shown in Figure 4.3.

```
testbed {--<flag>|<test>}

where <test> = 1,2,3 ...
<flag> is one of
    --verbose=<level>   Controls verbosity of the testbed itself
                        (JMTk verbosity is set via -X:gc:verbose)
    --time[=file]       Outputs timing information (appends to a file)
    --run=<n>           Number of runs for timing (records lowest
                        value)
    --skip=<n>          # runs to skip before starting to time
    --help              Prints a usage message
    -Xms<n>             Set the minimum heap size
    -Xmx<n>             Set the maximum heap size
    -X:gc:<flag>        Set a JMTk command line option
```

Figure 4.3: Testbed command line syntax

Tests are compiled in separate modules. Currently available test modules are:

`Allocate.c` Provides test functions that allocate memory with no garbage collection.

`Gc.c` Tests basic garbage collection functionality.

`FTree.c` Several variants on the 'functional tree' benchmark.

`QSort.c` Several variants on the 'quicksort' benchmark.

The main module assigns a number for each test (or to varieties of a test, if it takes parameters).

Two additional modules (and their associated header files) make up the testbed framework.

`object_model.c` Provides a programmatic interface to the testbed object model. In particular this provides functions to allocate objects with specified numbers of fields, and to get and set pointer and nonpointer values in the object. This module interfaces with the header file that defines the object model to JMTk.

```
/* From roots.h */
#define ENTER       int saved_sp = max_root
#define EXIT        max_root = saved_sp
#define PUSH(p)     root_table[++max_root] = (p)
#define LOCAL(id,v) int id = PUSH(v)
#define STACK(i)    root_table[i]

/* from FTree.c */
#define TREE STACK(tree)
void insertTree(int ret, int tree, int n) {
  ENTER;
  if( TREE == NULL ) Leaf(ret,n);
  else {
    LOCAL(left_i,Left(TREE));
    LOCAL(right_i,Right(TREE));

    if( n <= N(TREE) ) {
      LOCAL(new_left,NULL);
      insertTree(new_left,left_i,n);
      Branch(ret,new_left,right_i,N(TREE));
    } else {
      LOCAL(new_right,NULL);
      insertTree(new_right,right_i,n);
      Branch(ret,left_i,new_right,N(TREE));
    }
  }
  EXIT;
}
```

Figure 4.4: Testbed programming style: insert into a binary tree

`roots.c` Encapsulates the root set. The roots are held as a static array of `void *`, with a global counter pointing to the maximum live root pointer. The header file `roots.h` provides a set of macros that facilitate using the root array as a stack, some of which are illustrated in Figure 4.4.

The testbed programming style is somewhat artificial, because of the requirement that all JMTk heap pointers are stably stored in the root array across points where garbage collection may occur. Passing heap pointers as parameters and function return values is an extremely fragile operation, so in practice functions pass indexes into the root set instead of actual root pointers. Figure 4.4 shows the 'insert' function from the FTree benchmark[4]

The macros ENTER and EXIT take care of saving and restoring the stack pointer (maximum valid root pointer) value. The figure simplifies it slightly; actually both macros performs a bounds check—stack overflow can ocur as a result of some heap corruption errors. Instead of declaring local variables with heap pointer values, the

[4]described in detail in Chapter 8.

testbed functions declare local integer variables with the index of the root table element containing the pointer value, and the `LOCAL` macro facilitates this. When a function is called that returns a heap pointer, space must first be created on the stack for the return value, by pushing a NULL onto the stack. `Leaf` and `Branch` are macros that allocate an appropriately structured heap object, while `N`, `Left` and `Right` are macros that extract fields from a tree node.

4.4.4 Testbed object model

The object model is loosely based on the ghc object model; objects consist of a fixed header followed by 0 or more pointers, and 0 or more 32-bit non-pointers. While in ghc the header word is a pointer into an info table describing the object, in the testbed the word describes the object directly, using 15 bits each for the number of pointer and non-pointer words, and 2 bits for the use of the garbage collector.

Conditional compilation flags in the testbed source allow for 2 variants to this object model. For copying collectors, the forwarding pointer can either be written over the first non-header word in the object, or given a separate field in the header (`#define FW_PTR_IN_HEADER`). For debugging purposes, each header word can be surrounded by a buffer word, initialized to zero and periodically checked for integrity (`#define GUARD_WORDS`).

Other conditional compilation flags control the allocator in use, whether the GC requires a write barrier, and some optimizations on the C side of the language boundary. These options are discussed in detail in Chapter 8.

4.4.5 Testbed futures

The testbed has proven useful for the originally intended purpose of debugging the portable JMTk, and also in performance tuning and evaluation of the system. The testbed will also be useful for debugging new JMTk functionality because of the debugging support it provides through tools like gdb and gprof.

The testbed currently only provides two significantly complex tests, both of which are targeted at simple tracing collectors. These tests do no significant mutation of already built heap structures, and would serve as poor tests of generational and reference counting collectors. More complex tests will certainly be developed in the future.

While performance tuning and evaluation has been performed, it is well known in the literature that simple synthetic benchmarks are not good indicators of performance in real programs. The testbed programming model makes it difficult to implement real programs in the testbed, so one possibility for further performance work would be a trace-based tool that could replay the memory management behaviour of a real application. This could be a valuable tool for debugging errors in other environments.

4.5 Summary

This chapter has described the architecture and construction of the JMTk port and the testbed environment. The key design elements are: abstraction across a wide range of virtual machine/language runtime implementations; cross-language efficiency issues; and programming effectively with an accurate garbage collector in a language environment that does not support it.

This chapter outlined the new interface 'glue' and support environment that interfaces with the gcj-compiled JMTk code. In order to compile JMTk with the gcj compiler, the remaining Jikes RVM idioms needed to be turned into a form that an ahead-of-time compiler could work with, and this is the role of the source transformation tool described in the next chapter.

Chapter 5

Transform tool

The idiomatic Java used by Jikes RVM, where machine addresses are described by objects that are magically compiled to their correct representation, cannot be handled directly by a compiler for standard Java like gcj. The solution to this problem is to systematically transform the JMTk source code into a form that the gcj compiler will accept, and this chapter describes the tool that was developed to perform this transformation. This chapter describes the role of the transform tool in the JMTk port, how the tool was constructed, what its capabilities are and some of the possible uses the tool could be put to in the future.

5.1 Background

The idiomatic Java in which JMTk is written is described in detail in Chapter 3, and makes use of special features of Jikes RVM to represent low-level datatypes as Java objects. The canonical example is the `VM_Address` class, which represents an address in the underlying machine architecture. From the perspective of a programmer in Jikes RVM, a `VM_Address` is a slightly restricted kind of object, with methods for address arithmetic and conversion to and from the types `Object` and `int`. When the Jikes RVM compiler processes objects of this class, they are represented as address words of the appropriate length, and the methods are compiled to machine code sequences for address arithmetic etc.

The Jikes RVM approach is possible because the project 'owns' the Java Virtual Machine. In this project we don't have that luxury, and so the JMTk source has been modified in a systematic way to transform it into something that a vanilla Java compiler would accept. Note this is not a noe-time transformation producing a separate code base to the Jikes RVM version of JMTk, but one that is simply applied as a stage in the compilation process.

The required transformation is:

- Objects to `ints`. The various machine word types, `VM_Address`, `VM_Word` etc. are transformed to `int` or `long` depending on the address width of the target architecture.

- Instance methods to static methods. An `int` in Java cannot have a method, and so the instance methods of the types transformed above must be transformed to some other construct that will perform the required operation. As an initial step, they were transformed to static methods of a class corresponding to the original type. For example, the code sequence

```
VM_Address x = some_method();
other_method(x.add(4));
```

becomes

```
int x = some_method();
other_method(VM_Address.add(x,4));
```

where `VM_Address.add(a,b)` adds an offset to an address.

- Pragma exceptions are deleted. The Jikes RVM idiom uses phantom exceptions of the form

  ```
  some_method() throws VM_Pragma<xyz>
  ```

  and the calling code does not arrange to catch these exceptions. These exceptions are treated as directions to the Jikes RVM compiler as to how to generate code for the method. The transformed code drops these exceptions, and must drop the `throws` keyword if the resulting exception list is empty. It is also desirable to drop the corresponding `import` statement rather than provide a dummy class simply to satisfy the Java compiler.

- Renaming methods. Method names in Java classes can be overloaded, provided the overloaded methods take different parameter types. JMTk contains at least one class (the `Log` class for instance) that has more than one method with a magic machine-word type. After the transformation of types described above, all of these methods take `int` parameters, and violate Java's overloading restrictions. The solution is to change the name of the inctance to reflect it's pre-transformation type.

Using the source transform tool allows JMTk to become portable, without maintaining two separate code bases, and also opens up some future opportunities identified at the end of the chapter.

The instance method to static method transformation described above merits further explanation, as it is critical to the port. The interface layer for the gcj port provides an implementation of all the methods of each of the magic classes (`VM_Address`, `VM_Word` etc.), but as static methods. For example, the Jikes RVM `VM_Address` class has

```
class VM_Address {
  private int value;      // Magic compiler optimizes this away !
  ...
  VM_Address add(VM_Offset x) {
    // Intercepted by compiler magic; returns value + x
  }
```

```
  ...
}
```

and the gcj interface provides

```
class VM_Address {
  ...
  static int add(int a; int x) {
    return a+x;
  }
  ...
}
```

leaving the source transformer to translate the instance method calls to the appropriate static method calls.

5.2 Capabilities

The Jikes RVM Java idiom and the transformation process used to produce portable code was described above. The transform tool has been designed so that it can perform the required transformation, but in a way that it can easily be extended to cover future requirements. Several features were implemented that while not strictly required were easy to build and which have proven useful.

The transform tool has been implemented as a command-line tool, which requires a specification file as input. The specification file format is as follows:

```
package <old-package> => <new-package>;
```

Replaces the specified package name. Not strictly required, but helps by simplifying the 'build tree' and identifying whether the current source has been transformed or not.

```
type <old-class> => <new-class>;
```

Replaces the specified class name wherever it occurs in type-casts, variable declarations, **`instanceof`** expressions or method declarations. This is used to convert `VM_Address` etc to `int` types. Static method calls are replaced separately via 'method' transform specifications (described below).

```
import <old-import> => <new-import>;
```

Replaces the specified import. One or other of `<old-import>` and `<new-import>` can be blank, in which case the tool will add or delete an import specification respectively. If the class-name component of the import changes, this automatically triggers the corresponding type transform. This is not strictly required, but provides additional flexibility, and has proven useful to perform some one-off refactoring of the JMTk code.

```
prohibit import "<pattern>";
```

Prints an error message if the given pattern is found among the imports after transformations are complete. This is used in the JMTk regression testing to check for direct imports of Jikes RVM classes that bypass the `vmInterface` package.

```
method Class.name([type] param, ...) =>
       (Class'|param).name'(param, ...);
```

Transform a static method invocation. The method can be a static method of the same or another class, or an instance method of one of the parameters. If types are specified in the method signature, then the transform will only apply to a method with a compatible type signature; parameters with no type specification will match actual parameters of any type.

```
method (Class var).name([type] param, ...) =>
       (var|Class'|param).name'(param, ...);
```

Transform an instance method. Any parameter or instance variable name on the left-hand side can appear as the instance variable or a parameter on the right hand side, and a static class name can appear instead of an instance variable on either side. Where parameter types are declared, the method transform only applies to matching occurences of the method.

Some examples might make this clearer:

```
method (VM_Address a).add(o) => VM_Address.add(a,o);
```

`VM_Address.add` becomes a static method, with the address and offset as parameters.

```
method Log.writeln(VM_Address a) => Log.writelnAddress(a);
```

performs a simple rename of a method.

```
method VM_Magic.getLong(VM_Address a) => a.getLong();
```

Transforms a static method to an instance method of its argument (this example doesn't actually appear in the system implemented, but would, for example, be useful if JMTk were ported to the OpenVM system)[1].

The full syntax of the transform specification file is described in Appendix A.

5.3 Construction

Because transformations are applied based on the type of a method call and are required to swap instance variables for parameters etc, the transform utility needs to

[1]The port to OVM is actually underway at Purdue, and is using this source transformer.

analyse the source program beyond the simple syntactic level. Clearly a simple filter based on `awk` or `sed` would be inadequate for the task.

One very desirable feature of the transform tool is that it should presrve white space and line numbering as much as possible. This is important, because if compiler errors (for example) are reported when processing transformed code, the corresponding source can be located simply. Preserving comments also assists when locating errors and executing code in the debugger.

After considering a variety of options, the Antlr parser generator [Parr 2002; Parr and Quong 1995] was chosen, and this helped considerably in the development of the tool.

5.3.1 Antlr

Antlr is a parser generator tool in the tradition of Lex, Yacc, Bison etc, which provides the ability to generate Lexical Analysers, Parsers and Tree Parsers which traverse the generated abstract syntax trees [Parr 1994]. Generation of abstract syntax trees (ASTs) is performed in Antlr by a simple annotation of the parser rules, for example:

```
throwsClause
    :   "throws"^ identifier ( COMMA identifier )* ;
```

is the parser rule for recognizing `throws` clauses. This produces a syntax tree with a `throws` token at the root (specified by the postfix caret operator) and the comma separated list of exceptions as children.

The equivalent specification in the Antlr tree-parser language is as follows:

```
throwsClause
    :   #( "throws" identifier (COMMA identifier)* ) ;
```

where the notation `#(A B C)` recognises a tree with token A at the root and B and C as children. Both the parser and tree-parser can be annotated with code as is usual in such tools, and in the transform tool, the tree parsers are annotated whereas the parser is not.

The features of Antlr that made it suitable for developing this tool are:

- It is available in the public domain—there are no licensing issues.
- It is based on and generates Java and so does not increase the number of languages required to build JMTk.
- In addition to lexical analysers and parsers, Antlr builds tree parsers which automate the process of traversing a generated syntax tree.
- Antlr supports parsers that preserve white space by attaching hidden tokens to the syntactically significant tokens during the lexical analysis phase.
- A grammar for Java is distributed with Antlr.

- Inheritance of grammars is supported, so that the three phases of the transform tool can be coded "by difference" from a generic tree parser that simply traverses a syntax tree in the correct order.

The Java grammer supplied with Antlr is designed for applications which process the semantic content of Java programs, and as such it discards purely syntactic features such as semicolons, and builds identical parse trees for multiple variables declared in the same statement, and the equivalent declarations split over multiple statements. A modified parser grammar and tree-parser grammar were produced which captured all syntactic information originally present in the source code.

Antlr uses a factory pattern to produce both the token stream emitted by the lexical analyser and AST generator, and Antlr provides a subclass of its token and tree classes which allow white-space tokens to be hidden (not processed by downstream parsers), but chained between the visible tokens. In the final transformer tool, the AST is constructed from a subclass of this Antlr-supplied class that allows for line numbers and symbol tables to be added to each node of the tree.

Antlr also supports inheritance among grammars, and this feature is exploited in the construction of the transform tool. The majority of the processing of the tool is performed in 3 tree-parser grammars, which all inherit from a commmon ancestor. The base tree-parser grammar simply traverses the AST and calls an empty method on each token, in the order that they appear in the program source.

The first of the child grammars annotates the AST with symbol tables, producing an output tree where each symbol in the source code is mapped to its type. The second child grammar applies a set of transformations to the AST, and the third overrides the empty method with one that prints the tree with the intervening white-space tokens. The resulting program requires three passes over the AST to produce a result, but this is fast enough to be usable, and the corresponding improvement in clarity makes the inefficiency worthwhile.

5.3.2 Type information

In the Java programming language, the types of all variables cannot be determined without also reading and parsing the super-classes of the current class, all interfaces implemented by the class and all its super-classes, all classes explicitly imported by the source file, and other classes in the same package referred to in the program text. Java compilers will read both '`.class`' files and '`.java`' files in order to gather this information. In the interest of simplicity, the transform tool as currently written only deals with Java source code, and so cannot determine type information for classes such as those supplied by the Java run-time libraries. In practice this is not a significant limitation since JMTk rarely uses library classes (most of which allocate heap memory and so cannot be used in a garbage collector).

One significant implication of the decision to deal purely with Java source code is that the existing Java grammar cannot parse the conditional compilation used by Jikes RVM. Conditional compilation in Jikes RVM is processed using a C++ pre-processor which produces standard Java source from a set of symbol bindings. There are several alternate approaches, but the approach chosen was simply not to accept code with conditional compilation directives.

While the core JMTk package is free of these constructs, the Jikes RVM `vmInterface` package was not. The approach chosen has been to create an abstract `vmInterface` package purely for the purposes of transforming the program. This abstract interface package is syntactically correct Java to the shallow level required for parsing by the transform tool, but (for example) permits abstract static methods. This interface package serves to separate the interface definition from any concrete implementation, working around one of the limitations discussed in Chapter 9.

One design goal for the transform tool is that it should be able to work from and reproduce the existing directory structure of the JMTk package, and should in theory be able to transform the entire package in one invocation. In common with Jikes RVM, JMTk does not keep its source files in directories that correspond to Java package names[2]. The transform tool therefore needs to be told where to look for source files, possibly relative to the location of the current source file. One parameter to the tool is a list of directories to search for source files, which can contain a mixture of absolute and relative path names. Relative path names are interpreted relative to the source file currently being transformed.

The command-line syntax of the tool is given in Appendix A.

5.4 Limitations

The principal remaining limitation arises because the transform tool only operates on the level of Java source code. When the transform tool encounters a Java system class (eg `java.lang.Object` or `java.lang.String`), it cannot find the return types of its methods because the Java source is not available.

The second, less serious limitation is that it cannot parse code that uses the Jikes RVM conditional compilation syntax as described above. Parsing of code with conditional compilation that can occur arbitrarily is clearly beyond the capabilities of Antlr[3]. This limitation manifests in two ways: less seriously, the tool cannot transform code that uses conditional compilation, which is not relevant because the JMTk core package does not use this feature; but more seriously the tool cannot extract type information from source files which use this feature even if they are not being transformed.

The solution described above in Section 5.3.2 addresses both problems. In the future this could help address some of the limitations of Java as dicussed in Chapter 9.

[2]JMTk is due to be refactored into packages that will correspond to the current directory structure, and eliminate the need to shuffle files between directories at build time.

[3]See for example [Garrido and Johnson 2003] for one approach to refactoring C in the face of conditional compilation as an indication of the attendant complexities.

5.5 Future developments

Program transformation (both at source code and byte-code levels) is a powerful technique, and much more could be done with the tool than is currently implemented, especially to extract the best performance out of the gcj compiler. Specific enhancements could be:

1. Substitution of arbitrary expressions for method calls, effectively inlining simple method bodies at the call site. The current transformation replaces

   ```
   addr.add(offset)
   ```

   with

   ```
   VM_Address.add(addr,offset)
   ```

   which returns the value `addr+offset`. A considerable gain in efficiency is realised if the final expression can be substituted directly for the initial instance method call. The performance impact of this is shown in Chapter 7.

2. Inlining of methods. This could be used to implement the `VM_PragmaInline` idiom from Jikes RVM in cases where no compiler support is available for user-directed inlining.

   The gcj compiler has limited support for cross-class inlining of methods. The compiler's decision whether or not to inline is based on internal heuristics, and there is no facility for the programmer to influence the compiler's decision [4]. It should be possible (if nontrivial) to implement source-level inlining, based on the explicit inlining instructions provided by the Jikes RVM idiom.

3. Specialisation of methods. The Java keyword `final` is used to indicate to the compiler that optimal code can be compiled because the method cannot be overriden by a sub-class. In the design of a toolkit such as JMTk, many methods are not declared final but are not overriden in practice. The transformer could produce code where all methods not overriden are declared final, and could specialize class hierarchies so that all methods were final.

4. 'Out-lining' of fast-path code. In Jikes RVM, maximum performance from the write barrier and allocator is achieved by inlining the frequently used code path directly into the user code. Given current compiler technology, this is impossible to do where the user code is in a different language, and Chapter 7 demonstrates the value of this optimization. It should be possible for the source transformer to generate code in different languages for a restricted subset of Java sufficient to inline the most important methods.

[4] There is a global parameter that controls the inlining threshold, that can encourage the compiler to inline larger methods. Encouraging more inlining across the board can actually be detrimental to performance, and there is still no way to override the compiler's decision about whether a *specific* method is a candidate for inlining.

5.6 Summary

This chapter has described the transform tool used to produce code that can be compiled by gcj from the existing Jikes RVM specific code base. Together with the work described in Chapter 4, this produced a working version of JMTk ready to be integrated into a programming language runtime system. The next chapter describes the work that was done to incorporate JMTk into the ghc runtime, and the reasons that work was not completed.

Chapter 6

GHC

The original target for the port of JMTk was ghc, the Glasgow Haskell Compiler. In practice, porting to ghc was too complex an undertaking to be completed in the time available. This chapter discusses the GHC runtime, the requirements it places on JMTk, the sources of complexity that led to it not being completed, and some thoughts on how to approach complex software in general.

6.1 The Spineless Tagless G-machine

The Glasgow Haskell Compiler compiles Haskell programs to a virtual machine described by Peyton Jones [1992], a virtual machine derived from the G-Machine [Johnsson 1984; Augustsson 1984; Peyton Jones 1987]. The G-Machine is a stack-based graph reduction engine, designed so that a program in a lazy functional language (Lazy ML, in the first case) can be compiled to G-code, and then either interpreted or compiled to machine code.

The Spineless Tagless G-machine (STG-machine) is an abstract machine for the execution of a simple typed lambda calculus, known in the compiler documentation as the 'core' language (the STG language in [Peyton Jones 1992])[1]. This language is formally specified by an operational semantics in terms of a state machine with 5 components, the code, the argument stack, the return stack, the update stack, the heap and the global environment. The paper also gives a translation from the abstract machine of the operational semantics into a more concrete machine, giving fragments of C code that correspond to steps in the operational semantics.

Heap objects in the STG-Machine are closures, representing either values or incomplete computations. Heap objects have a uniform layout, consisting of a header, followed by a number of pointer fields, and then a number of non-pointer fields. Compilation options (in particular certain profiling options) can add fields to a closure header, but usually an object header consists of a single pointer to an information table (the info-pointer) that describes the closure.

In particular, the information table contains pointers to code fragments that 'enter' the object (for function invocation), 'evacuate' and 'scavenge' the object (for copying

[1] Not actually 100% true—the core and STG languages differ slightly: see the paper for details

garbage collection). Copying garbage collection is described in the STG paper as a combination of 2 operations:

evacuate Copies the object from from-space to to-space and overwrites it with a forwarding pointer.

scavenge Traces the pointers in an object, and evacuates the object if it hasn't already been done, and then scavenges it.

Common object types (for example a closure with 1 pointer and 1 non-pointer field) share a common 'scavenge' function whether or not they are actually the same type, and objects without a specialised type share a generic 'scavenge' function.

The majority of the interface glue that JMTk requires in order to support ghc can be constructed by suitably modifying the existing evacuate and scavenge functions.

6.1.1 Full laziness

One important feature of the G-machine and its descendants is that they implement full laziness [Wadsworth 1971]. For example, in the expression

```
(λa.a × a) (sqrt 4)
```

a simplistic normal order evaluation strategy first reduces this to

```
(sqrt 4) × (sqrt 4)
```

causing `sqrt 4` to be evaluated twice. Full laziness guarantees that it will only be evaluated once, by substituting instead a 'pointer' to the expression `sqrt 4` when the lambda expression is reduced. This substitution has a straightforward representation in the graph that the G-machine produces to represent the computation.

When executing the program, ghc must be careful to preserve the semantics. A simplistic evaluation strategy will evaluate the first parameter of '×', form the result, 2, in a new heap cell, and update the pointer that led to this node to point to the result. Unfortunately the pointer to the second operand will still point to the original `sqrt 4` graph, and full laziness is violated. The G machine solves this by overwriting the `sqrt 4` node in the graph with an *indirection* pointer to the new result, recovering the fully lazy behaviour. These redundant indirection nodes are removed during garbage collection, as the collector traces the heap.

6.2 The GHC Runtime

The GHC runtime represents 12 or more years development over and above the original STG-Machine, and in a significantly complex piece of software, comprising almost 100,000 lines of C. Via conditional compilation, the GHC runtime supports at least 4 separate parallel/distributed execution models, making code such as the scheduler opaque in the extreme. The latest documentation is available in the STG Runtime Document [Peyton Jones et al. 1999].

The main difference in complexity, from a garbage collection point of view, over the original STG machine specification is the addition of concurrency. With this, stacks are allocated in the heap and both scavenged and evacuated by the garbage collector.

6.3 GHC challenges

Ghc requires two features that JMTk does not currently provide:

- Root discovery during collection. One component of the root set in ghc is the list of static objects. These are closures created by the compiler in the data segment of the generated executable, that represent top-level functions. Particularly where libraries are used, the full set can be very large, but only a few are in use at a given time. These objects come into scope as the graphs of functions are expanded during evaluation, and drop out of scope again as they become evaluated, replaced with closures that represent their partially or fully evaluated values. They differ from heap closures in the sense that they cannot be copied.

  At present, JMTk assumes that all root pointers can be identified before tracing the heap commences. To accommodate this feature, JMTk would need to allow for the root set to grow during tracing, and to loop over the set of root pointers, repeatedly tracing the heap until the root set is empty.

- Indirection nodes. As mentioned above, indirection nodes are removed by the garbage collector during the copy phase. During copying, indirection nodes are skipped, and are overwritten with forwarding pointers that point to the (copied) object that the indirection pointed to. JMTk does not currently have a mechanism for handling this kind of operation. Unfortunately the impact of this optimization on Haskell runtimes is not known.

Neither of these features would require fundamental changes to JMTk to implement, but the changes would not be trivial either.

6.4 Ghc and JMTk

Approximately 2 months was spent on integrating JMTk and ghc, and in late August ghc was running simple test programs, allocating memory from JMTk, but without garbage collection. Soon thereafter, garbage collection was enabled against an incomplete interface that was expected to fail—and fail it did, but in a completely unexpected section of the ghc runtime.

Close examination of the code where it failed showed that the modular allocation code (the ghc storage manager) corresponded essentially to the slow path of the ghc monotonic allocator, and that allocation of memory was in fact distributed much more widely through the runtime than was previously thought.

In retrospect the lack of modularity in the memory allocation code should have been expected: the G-machine specification talks explicitly in terms of heap and stack pointers, so manipulating them directly at the points where the virtual machine specification requires it was the most natural approach. Figure 6.1 shows part of the STG Machine definition and illustrates the point.

```
/* Allocate heap block */
Hp = Hp - 3;
if (Hp < HLimit)
        { ...trigger GC... }

/* Fill in closure for gx */
Hp[0] = &gx_info;
Hp[1] = SpA[1];
Hp[2] = SpA[2];
...
```

Figure 6.1: Part of the STG-Machine definition

The next step in integrating JMTk into ghc will be to refactor the runtime so that allocation code is modular, using C macros (where a bump-pointer allocator is used) to preserve performance.

6.5 Working with complex code

The problems encountered with ghc raise the question of how problems like this could be avoided in future exercises. With systems this large or larger, it is clearly impossible to read and understand the entire system before attempting to work with it, and attempting to drill down to only those modules relevant to the problem at hand, as was done here is clearly not sufficient.

I have concluded that an experimental approach should be taken, by instrumenting the code in question and observing its behaviour. In this case, printing a message at each allocation, and again as the garbage collector scanned each object would have uncovered objects that weren't allocated by the expected interface. Investigating complex code by treating any understanding gained by an inspection of source code as a hypothesis that needs to be verified by experimentation is probably a healthy approach to complex systems such as this.

6.6 Summary

Integrating ghc with JMTk remains an unfinished project. There is no reason why it could not be completed—work was ceased due to time constraints, not because the project is infeasible.

Abandoning the ghc integration allowed some time to perform the work that is the topic of the next chapter, performance tuning, and to develop the testbed so that it could be used for benchmarking, which forms the basis for the third part of the thesis.

Chapter 7

Tuning

The previous parts of this thesis discuss how the testbed environment was constructed, and the porting of JMTk was performed. The philosophy adopted was to concentrate on correctness, and leave performance tuning and optimization until after JMTk was running in the new environment. This chapter describes the steps taken to improve performance. It concentrates in detail on the steps taken to optimize the JMTk port in the gcj testbed environment.

7.1 Motivation

Initial performance results observed during functional testing were poor, and it was clear soon after the port was running in the testbed environment that work was needed to improve it. In the initial semi-space collector, collecting a 1MB (semi) heap was taking 1 second, which although there was little to compare this figure to, seemed much too high.

In order to establish a baseline for performance tuning, two variants of the testbed were implemented, firstly using a pure 'bump pointer' allocator, and secondly using the system-supplied 'malloc' function[1]. This chapter describes the steps taken to improve performance. Chapter 8 evaluates performance of the tuned collectors against each other and against other systems, and describes the benchmarks and benchmarking methodology.

Tuning was done progressively using the semispace collector and the functional tree benchmark described in Chapter 8. After each tuning step was performed, the program was tested at a range of heap sizes, and the results graphed to illustrate the effect of each optimization.

7.2 Optimization

The optimization process was directed by the analysis of profile results using the 'gprof' tool. Profiles were taken using the minimum and maximum heap sizes, to

[1] This was on Linux with glibc 2.2, so this is a recent version of the Lea allocator.

direct optimizations to the two different codepaths in JMTk. At each stage in the optimization process, an optimization was identified, applied, and results were graphed to show progress. The semiSpace collector with the FTree benchmark was used for this exercise.

In the discussion below, the term 'magic classes' and 'magic types' refers to the java classes created to implement operations on the types `VM_Address`, `VM_Word` etc, which are the unsigned int types that the Jikes RVM 'magic' compiler replaces with machine instructions.

1. Replacement of `VM_xxx` calls in Deque, LocalDeque and VMResource classes.

   The first two classes are the supertypes of a family of double-ended queues used extensively during garbage collection. The body of the methods consists mainly of arithmetic on various magic types, and their poor performance appeared to be due to the method invocations involved. The other class is fundamental to managing regions of virtual memory

   Replacement of the method invocations with the appropriate arithmetic operations was performed, effectively inlining the method calls at a source code level.

2. Inlining of `VM_xxx` methods in the BumpPointer class.

   This optimization pass was targeted at the allocation path, performing inlining as above.

3. This step is discussed in more detail below. It involved duplicating the java code that represents the 'fast path' of the bump pointer allocator into the C++ interface code, replacing the call `TB_Interface::allocate(bytes)` with the function[2]

```
char *buf, *bufEnd;
char *bp_alloc(int bytes) {
  if( buf + bytes > bufEnd ) {
    buf = TB_Interface::allocate(CHUNK);
    bufEnd = buf + CHUNK;
  }
  buf += bytes;
  return buf - bytes;
}
```

   In the frequent case where small objects are allocated from the pre-allocated buffer, this code sequence allocates memory in an almost ideal number of instructions.

4. The latest version of gcj provides the ability to perform limited cross-class inlining, provided the application is compiled on a single command line.

[2]The actual function is more complex than this, but this gives the essential idea.

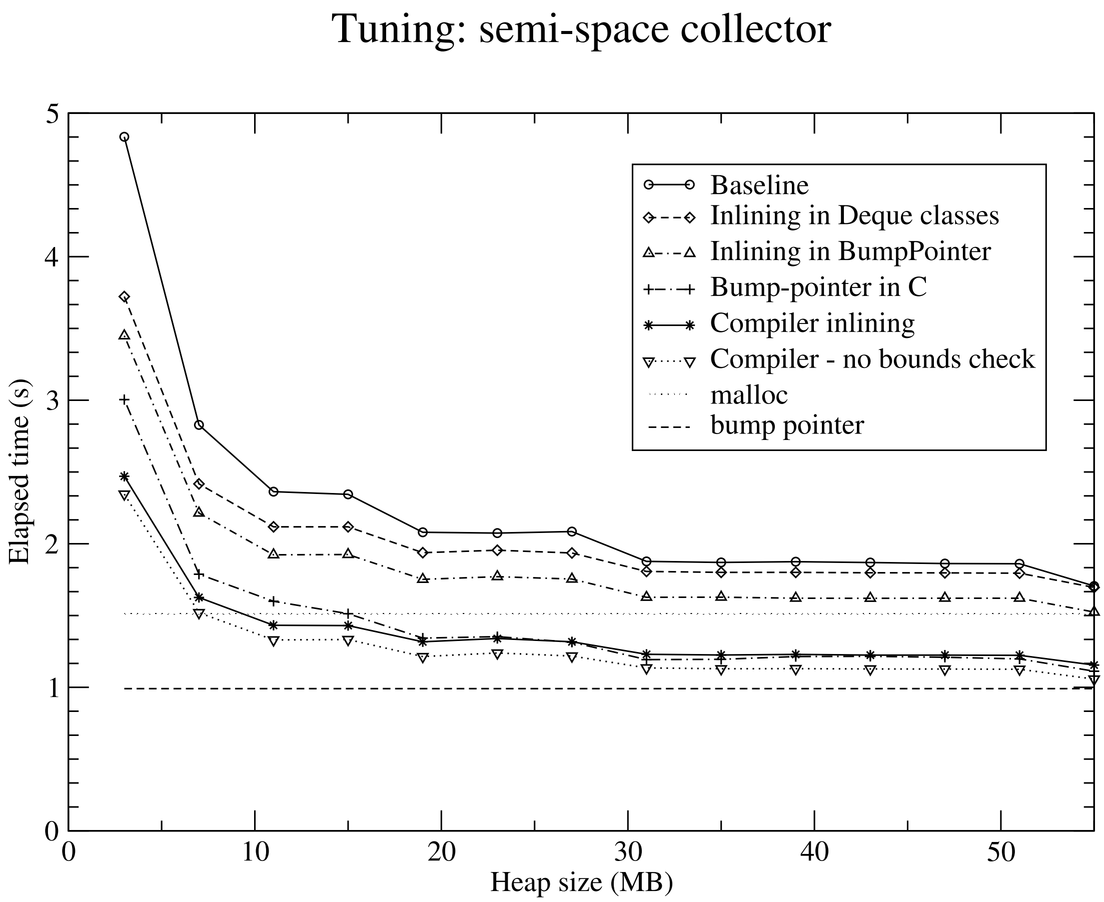


Figure 7.1: Tuning results

5. One further optimization optionally available in gcj is the elimination of bounds checking on array accesses.

The net result of these optimizations was a 51% speedup in the smallest heap size, and a 38% speedup in the largest. These results are presented graphically in Figure 7.1.

7.2.1 Other collectors

The performance bottlenecks in other collectors, particularly the mark-sweep hybrids (and, when they are functional, the reference counting collectors) are in different classes. The same methods have been applied to these classes, but the process was not analysed in as much detail. Of particular note in the segregated free-list class which is used for allocation by the mark-sweep space was the `VM_AddressArray` class (one of the 'magic' classes in Jikes RVM). This is implemented in JMTk by an object encapsulating an array. Direct inlining of an array produced significant improvements even in the face of compiler inlining.

Moving the frequent path of the free-list allocator into C code would produce significant gains for the pure mark-sweep collector, but would prove much more difficult than inlining the bump-pointer. Comparing the performance (as in the next chapter) of the Boehm collector and the JMTk mark-sweep collector shows how much scope for improvement there remains.

7.2.2 Evaluation of optimization strategies

Taken as a whole, the optimizations performed fall into 3 categories: fixes to the gcj interface code base; inlining of the methods used to transform Jikes RVM idioms to gcj-compilable code; and 'functional inlining' of code across language boundaries. By far the most important are the latter two.

Hand optimization of code produced by an automatic transformation is clearly a suboptimal strategy. Development of the ability to transform method calls into arbitrary expressions in terms of the method arguments would address the issue, and carry the performance improvements across to other JMTk components. This idea could not be implemented in the time available, and was described in Chapter 5. In the meantime, the optimizations have been turned into a patch that is applied during the build process of the ported system.

Hand inlining would seem to be unnecessary when compiler inlining is available, however this was confirmed not to be the case. Using gcj inlining as the first step in the optimization process produced similar percentage gains.

Inlining of the 'bump pointer' allocator into the C code was identified very early in the project as a potentially fruitful optimization. Various authors (e.g. [Appel 1989]) discuss the performance of bump-pointer allocators, and indeed it is one of the often cited advantages of copying garbage collection algorithms. A similar approach has been implemented for write barriers for the generational collectors. This particular optimization is not one that could be implemented automatically, although a form of support could be provided as described in Chapter 5.

Implementation of this optimization has been kept as conditionally compiled code, as it cannot be applied to memory management plans that require free-list allocation such as a pure mark-sweep or reference counter collectors. It would be valuable for JMTk to provide an interface call on initialization that would check whether the plan uses a bump-pointer for allocation so as to prevent mismatches between the C and Java portions of the runtime. There is also the issue of plan-specific post-allocation initialization to be addressed, by exposing the plan's post-allocation initialization function in the interface.

7.3 Future work

Tuning of the gcj port of JMTk is far from complete. While performance of the testbed in allocation-intensive situations (in large heap sizes) approaches the ideal, collection performance is still poor as the performance results of the next chapter show.

The following strategies are likely to improve performance:

1. The source transform enhancements discussed in section 5.5 are all aimed at

enhancing performance. The hand-tuning described in this chapter shows the value of item 1.

2. Close examination of compiler inlining. gcj can indicate where its inlining process fails, and it may be possible to direct the inlining process further. It may be possible to encourage the inlining process using hints from the Jikes RVM pragma that is currently discarded.

3. Exposing more of the internals of the most performance sensitive parts of JMTk, for example the bump-pointer and free-list allocators. For example, currently the size of the 'cache' is constrained because the collector allocates objects of $>$ 4K in the large object space. The JMTk plan could provide a method for the caching code to allocate its buffer directly, bypassing this rule.

7.4 Summary

The effort expended on tuning the ported JMTk system consumed relatively little of the time spent on the project, yet it has produced a significant improvement in performance. Some features of the port which were expected to limit performance did not do so, and the 'build-first, tune later' approach was vindicated.

This concludes part 2 of the thesis, and has described the process by which a portable version of JMTk was constructed. The final part of the thesis evaluates the performance of the final product, and draws some conclusions.

Part III

Results and conclusions

Chapter 8

Performance

The previous parts of this thesis described the research goals and problems to be solved in this project, and described in detail the construction of the end result: a portable version of the JMTk memory management toolkit. The third and final part of this thesis evaluates the system that was built, and the role of this chapter is to evaluate the performance of the system. It discusses the programs used to benchmark the system, presents results comparing the portable JMTk with the Boehm collector and other systems, and then looks at future possibilities for benchmarking in the testbed environment.

8.1 Introduction

Evaluating the success of the porting project primarily involves evaluating the performance of the final product. If the original aim of building JMTk into the ghc language runtime had succeeded, this would have been simple: run the same set of benchmarks in the old and new collectors and compare the results. Finding a fair basis of comparison for the testbed environment is a little more problematic.

The testbed programming environment described in Chapter 4—where accurate garbage collection is performed without compiler support—limits the choice of benchmark somewhat, and the benchmarks implemented are discussed in the next section of this chapter.

One immediately available basis for comparison is the Boehm collector. By trivially substituting a call to `GC_MALLOC` (the Boehm collector's allocation function) for the call to the JMTk allocator, we can run any benchmark under the Boehm collector. This gives a comparison using the same mutator, so the only performance difference is the memory manager.

At the time of writing, four of JMTk's seven collectors were running successfully, and a comparison of those collectors in the testbed has been performed.

Another interesting comparison, although the end results require careful interpretation, is between the testbed and the same benchmarks (in memory behaviour terms) running in other Java virtual machines. This comparison is given in Section 8.3.2.

Benchmarking of garbage collectors is a very problematic area, as the performance characteristics of different collectors are very sensitive to nuances in program behaviour. The final part of this chapter discusses how some of the weaknesses of the benchmarking procedure applied here could be addressed in the future.

8.2 Benchmarking

The restrictive programming model of the testbed environment, in particular the requirement that all heap pointers be stably held in the root pointer array dictates that any benchmark be a simple artificial benchmark (alternative approaches that may lift this restriction are discussed in Section 8.5).

Two benchmarks were implemented in the testbed environment, chosen because: they performed significant memory allocation; were relatively simple to implement; and yet resembled—if not real applications— algorithms that might be found in real programs.

8.2.1 The functional tree benchmark

The first benchmark program is a persistent binary tree implementation (also referred to here as a functional tree), where insertion and deletion are performed by allocating duplicate nodes from the root of the tree to the modified node in the tree. This type of data structure is common in the functional programming world, where both new and old trees need to be visible to the programmer, in order to achieve referential transparency. The insert operation is illustrated in Figure 8.1, and the algorithm used for insert and delete operations is illustrated in Figure 8.2.1.

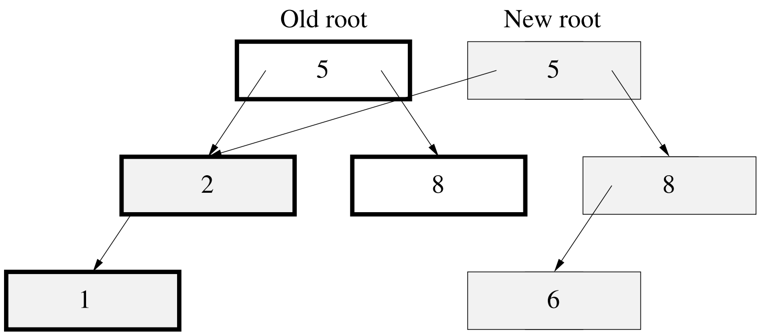


Figure 8.1: Insert into a persistent binary tree. The nodes in the initial tree are bolded, and the nodes in the result tree are shaded. The un-shaded nodes will become garbage when all pointers to the old root disappear.

The benchmark program chooses integers from a pseudo-random stream, inserting them into a tree until the tree reaches a given size. The benchmark then alternately inserts and deletes values from the tree. The size of the initial tree and the number of values inserted and deleted parameterize the benchmark allowing its runtime and heap occupancy to be tuned. For the benchmark runs described in this chapter, a tree of 20,000 nodes was built, and then 10,000 insert and delete operations were performed on the final tree. The benchmark is known as 'FTree'.

```
data Tree = Node Tree Tree Int | Null

insert v Null = Node Null Null v
insert v (Node l r v') =
    if v <= v' then Node (insert v l) r v'
               else Node l (insert v r) v'
delete v Null = Null
delete v (Node l r v')
    | v < v' = Node (delete v l) r v'
    | v > v' = Node l (delete v r) v'
    | v == v' && l == Null = r
    | _ = let v'' = findMax l
          in Node (delete v'' l) r v''
findMax (Node l Null v) = v
findMax (Node _ r _) = findMax r
```

Figure 8.2: Core functions of the Functional Tree benchmark, in pseudo-Haskell. Each application of the `Node` constructor causes memory to be allocated.

Two variations of this benchmark were studied, with additional 'payloads' in each object allocated, consisting of 40 or 80 bytes of non-pointer data.

8.2.2 The quicksort benchmark

The second benchmark is a list-based quicksort program. As with the FTree benchmark, a functional style is adopted, by duplicating lists in the 'partition' step. The benchmark is given in pseudocode in Figure 8.3.

```
qsort [] = []
qsort (p:xs) = let (lt,ge) = partition p xs
               in append (qsort lt) p:(qsort gt)

-- This is iterative, to reduce mutator cost
partition p xs =
    let (lt,ge) = ([],[]);
    for x in xs
      if x < p then lt = x:lt      -- Memory is allocated when
               else ge = x:ge      -- constructing these lists
    return (lt,gt)

-- Append is actually iterative, and duplicates the first list
append [] ys = ys
append (x:xs) ys = x:append xs ys
```

Figure 8.3: Key functions of the quicksort benchmark.

The QSort benchmark consists of allocating and then sorting a series of lists of random integers, of lengths 1 through to 2^{16}, at steps of 2^n. Each list is allocated, sorted, and then checked to verify that it is in fact sorted and the same length as originally allocated.

8.2.3 Measurement

Measurements were performed by running the benchmark multiple times in a loop inside the test program, and taking the best time recorded. A garbage collection is requested after each iteration. This procedure ensures that startup times are not counted, and compilation costs (for the Java virtual machines measured) are neglected.

8.3 Results

Results of the performance tests are graphed in terms of elapsed time (best time of 5 iterations) at a range of different heap sizes. As the minimum heap size that each benchmark will run in varies from collector to collector, the leftmost point of each curve represents performance at the minimum heap size for that collector. The x-axis of the graph is given in terms of MB in excess of the minimum for each collector. For example, if the minimum heap for the semispace collector is 3MB, for gcj is 6MB and for Jiikes RVM is 15MB, the point ′8′ on the x-axis represents an absolute heap size of 11MB, 14MB and 23MB respectively[1]. The minimum heap size for all benchmarks run in the testbed is kept constant across collectors, and actually represents the minimum heap in which the semi-space collector would run. The x-range of the FTree(0) and QSort benchmarks were chosen so that no garbage collection was performed at the rightmost edge of the graph. The remaining FTree benchmarks were too large to do this in the main memory of the test platform, so they were graphed on the same scale as the base FTree benchmark.

8.3.1 Testbed performance

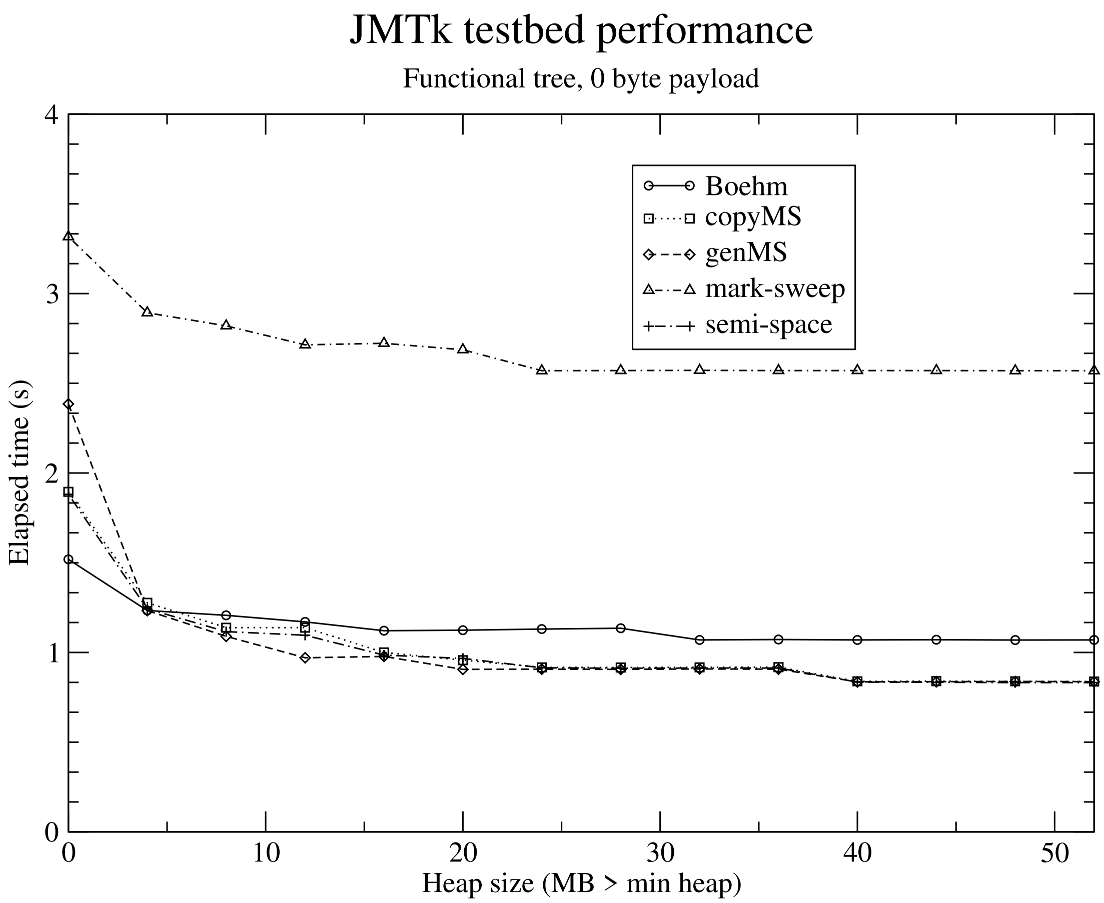
The performance of the testbed collectors was measured with 4 benchmarks: FTree with 0, 40 and 80 bytes of additional data[2], and QSort. Graphs of the benchmarks are given in Figures 8.4 through 8.7.

Things of interest to note from these graphs:

- Mark-sweep is consistently the worst performing collector. The mark-sweep collector is the only one to use JMTk's free-list allocator—the other JMTk collectors use a bump-pointer, part of which is 'inlined' into the C testbed code. The difference in cost for mark-sweep would therefore appear to be the cost

[1] Blackburn, Cheng, and McKinley [2004a] use multiples of the minimum heap size. I found that when comparing systems with different fixed overheads this gave collectors with high heap requirements an unrealistic advantage. For example comparing the Sun Java VM against Jikes RVM with 10 different heap sizes, the rightmost point on the graph would represent 10MB and 150MB respectively.

[2] The mean object size in Jikes RVM for the Spec Jvm98 benchmark suite is 32 bytes, and 98% of objects are 96 bytes or less. The objects used in these benchmarks are 16, 56 and 96 bytes respectively.



JMTk testbed performance
Functional tree, 0 byte payload
Boehm
copyMS
genMS
mark-sweep
semi-space
Elapsed time (s)
Heap size (MB > min heap)
0
1
2
3
4
0
10
20
30
40
50

Figure 8.4:

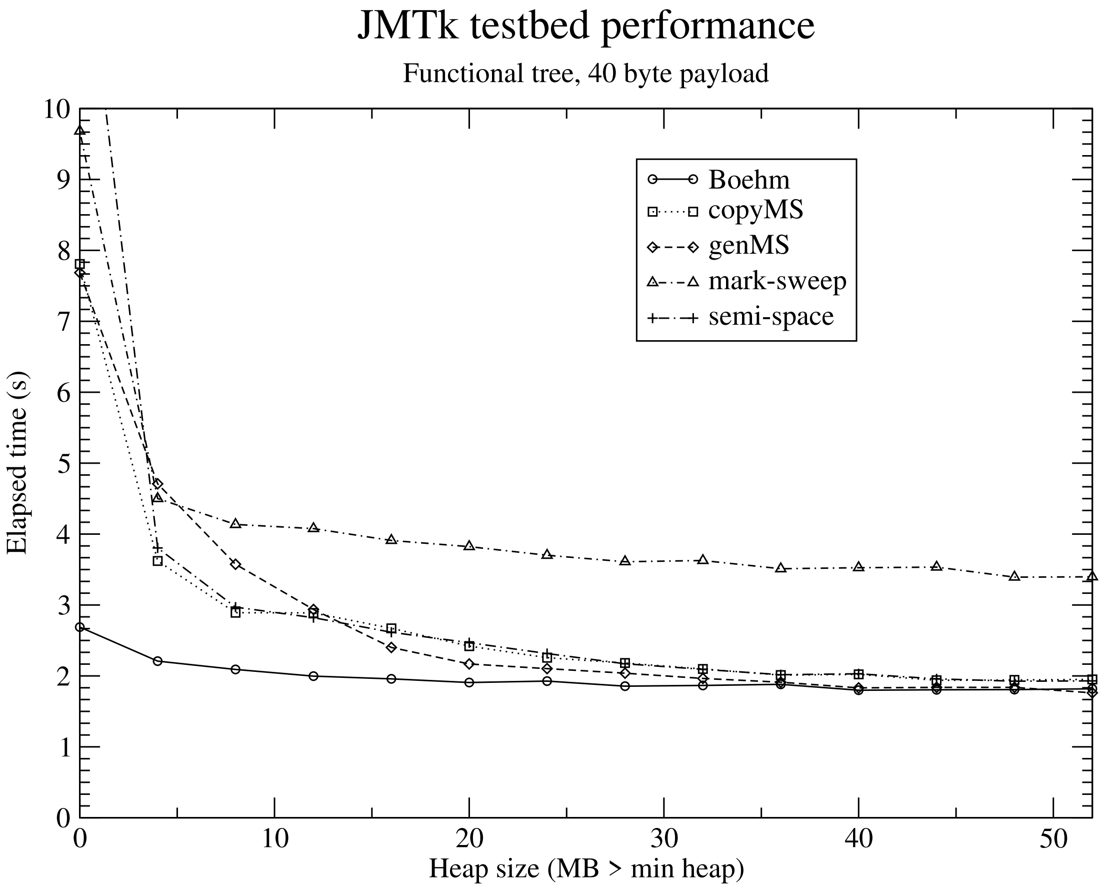
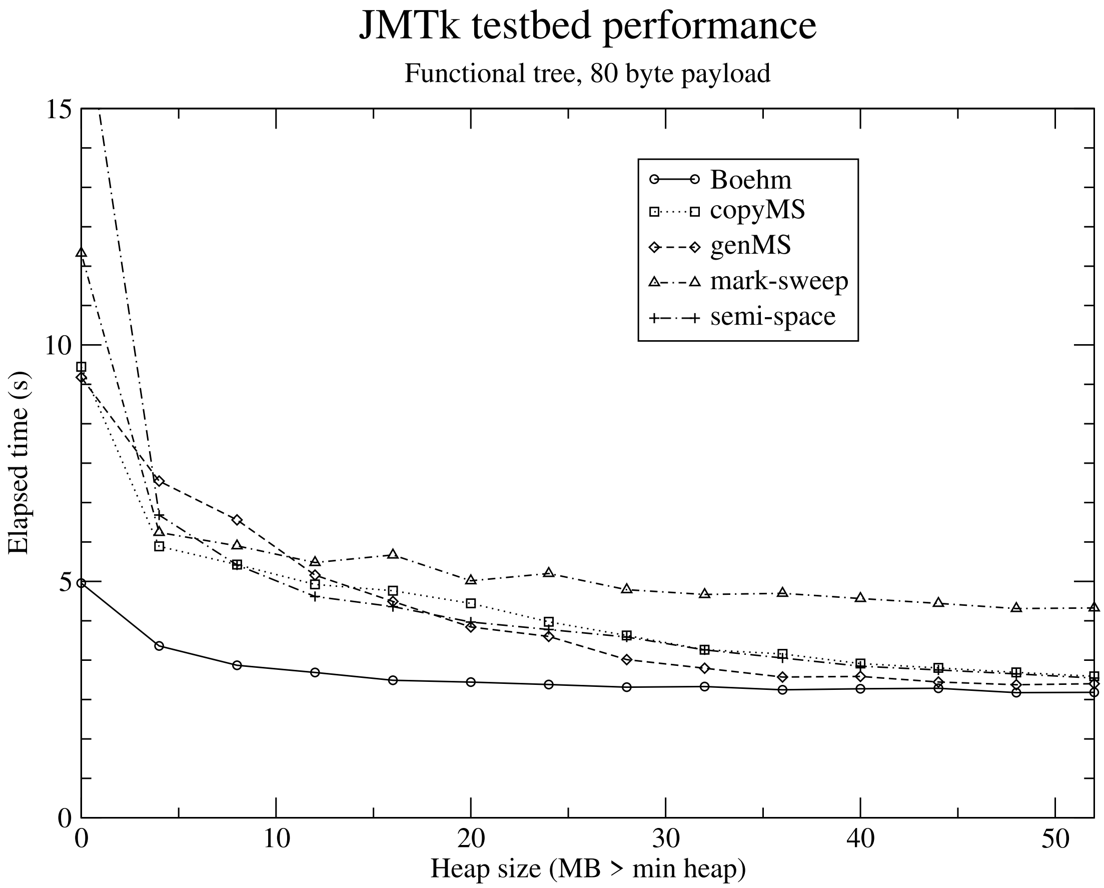
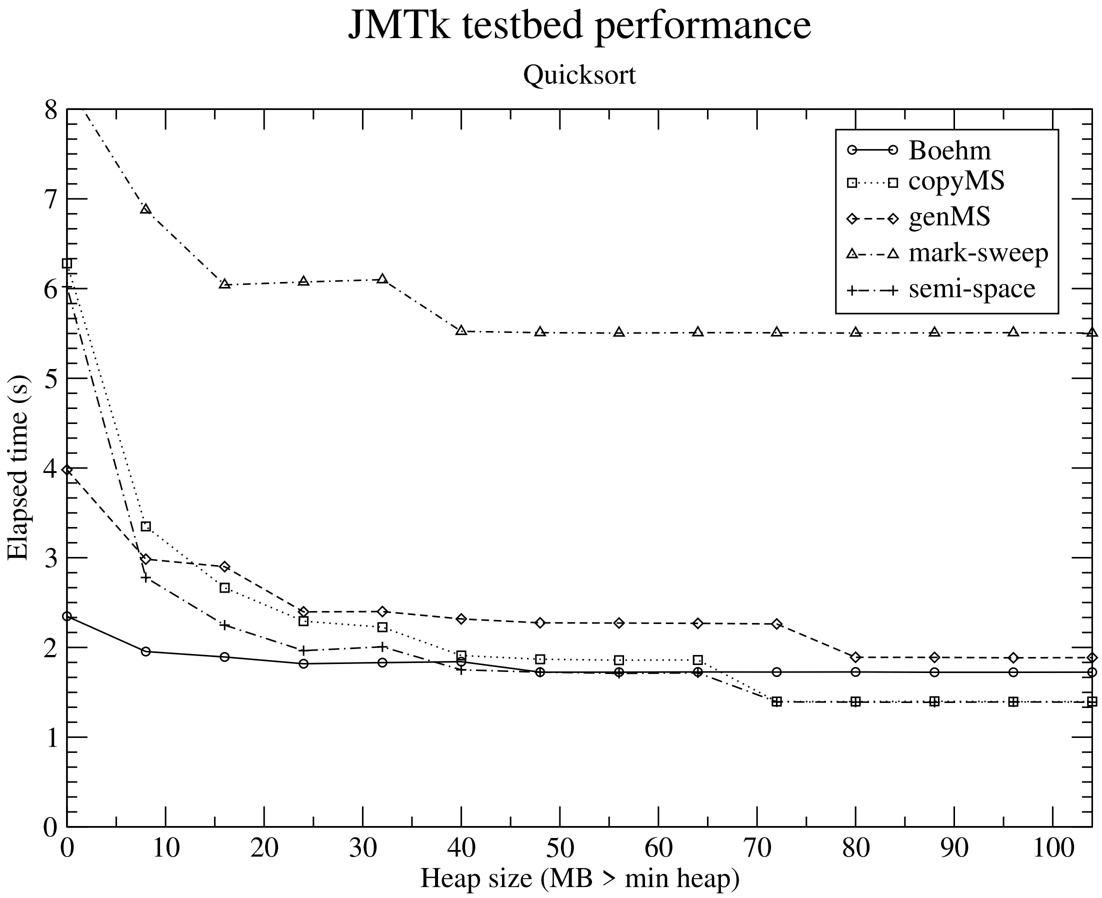

JMTk testbed performance
Functional tree, 40 byte payload
Boehm
copyMS
genMS
mark-sweep
semi-space
Elapsed time (s)
Heap size (MB > min heap)
0
1
2
3
4
5
6
7
8
9
10
0
10
20
30
40
50

Figure 8.5:



JMTk testbed performance
Functional tree, 80 byte payload
Elapsed time (s)
Heap size (MB > min heap)
Boehm
copyMS
genMS
mark-sweep
semi-space
0
5
10
15
0
10
20
30
40
50

Figure 8.6:



JMTk testbed performance
Quicksort
Elapsed time (s)
Heap size (MB > min heap)
Boehm
copyMS
genMS
mark-sweep
semi-space
0
1
2
3
4
5
6
7
8
0
10
20
30
40
50
60
70
80
90
100

Figure 8.7:

of crossing the language boundary into the Java code for each allocation. In Figure 8.4 and Figure 8.7, the performance of the mark-sweep collector is significantly worse than the other collectors, but the difference is less noticeable in the other graphs, which have larger object sizes. This is consistent with there being a per-object overhead which diminishes in significance as per-byte costs increase with the larger object size. The performance of the genMS collector, which allocates its mature space with the same class (especially visible in Figure 8.6) indicates there is nothing wrong with the performance of the free-list allocator when called from Java code.

The Boehm collector also uses a segregated free-list allocator, but optimized over the 16 years since the collector was first developed. It should be possible for the JMTk free-list allocator to achieve a similar level of performance.

- In small heaps, and where the object size is large, the Boehm collector outperforms the JMTk collectors. The relative performance of the Boehm collector is discussed in more detail in Section 8.4. With larger object sizes, the overhead of copying collectors is proportionally higher, and the mark-sweep algorithm used by the Boehm collector obtains more of an advantage. Also, the bump-pointer 'cache' of the JMTk collectors is less effective when objects are larger, and the slow allocation codepath is called more frequently. Performance improvements for this were discussed in Section 7.3.

- Performance of these collectors does not match the relative performance of the JMTk collectors in Jikes RVM as shown by Blackburn, Cheng, and McKinley [2004a]. One factor in this may be the choice of benchmark, but the results of the next section (where two collectors running in Jikes RVM are compared) don't seem to support that hypothesis.

  The most likely explanation is that in the tuning effort, the semi-space collector received more attention than any other collector, and there is still significant tuning to be done.

  Another, less welcome possibility is that semi-space performs well because it is a very simple collector. The generational collectors perform less well than in Jikes RVM because the gcj compiler is unable to perform the same amount of optimization as Jikes RVM, and that complex, well-structured (from an OO-design point of view) will not perform well under gcj. If this is true, then significant effort will need to be expended to make JMTk competitive with hand-tuned C implementations, and may requore all the strategies suggested in Chapters 5 and 7 as future possibilities.

In summary, performance of the JMTk collectors in the new environment is neither extremely good or bad. Work is required to improve it before compiler developers currently using the Boehm collector could be persuaded to implement the code necessary to perform accurate collection in order to interface with JMTk. On the other hand, the possible approaches to tuning have not yet been exhausted.

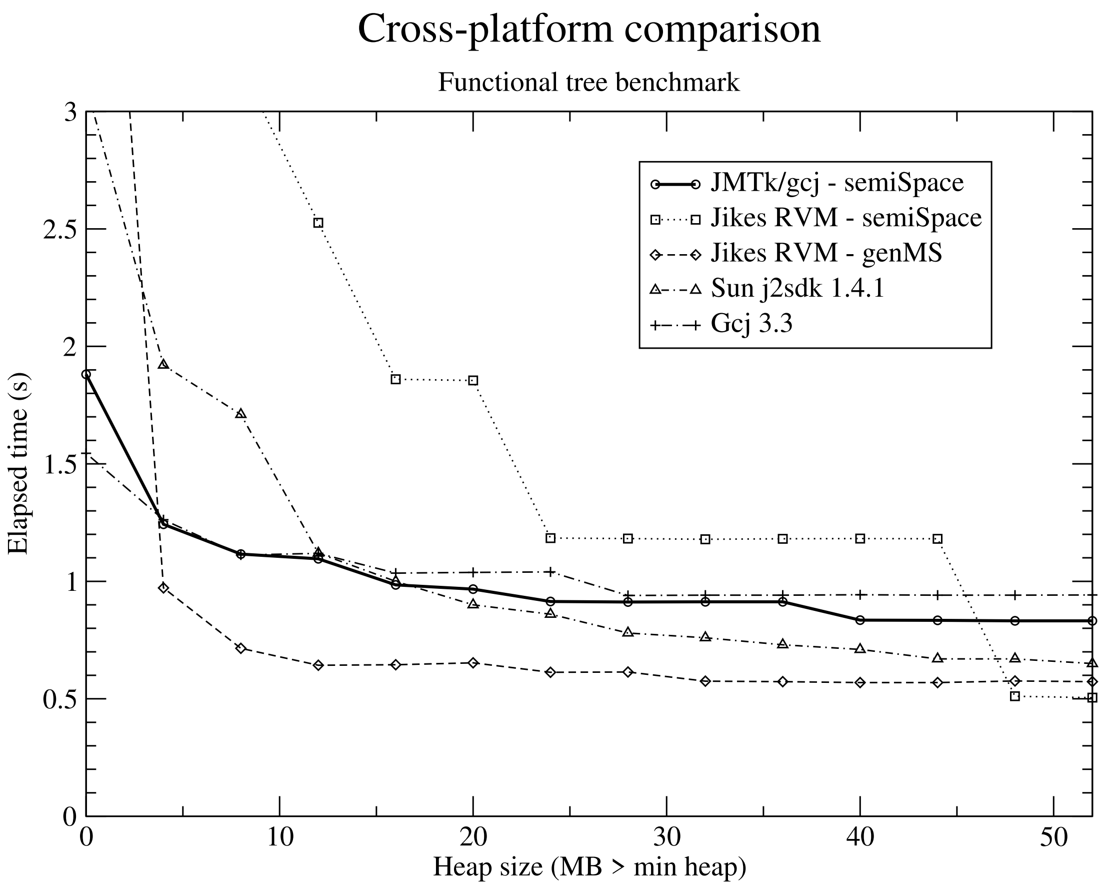


Figure 8.8: Cross platform performance comparison: the functional tree benchmark.

8.3.2 Cross platform

Figures 8.8 and 8.9 show the relative performance of the testbed with a semi-space collector and 4 other Java systems: Jikes RVM running the JMTk semi-space collector; Jikes RVM running the genMS collector; gcj 3.3 running the Boehm collector; and the hotspot Java virtual machine from the Sun Java SDK 1.4.1, which uses a generational copying collector.

The basis of comparison is problematic. While there are bases for comparison for all systems, there are also substantial differences. The comparison of the testbed with Jikes RVM seems on the face of it to be the most valid, as this is the exact same collector. On the other hand, while the heap in the testbed contains only the data structure of the benchmark, the Jikes RVM heap also contains considerable amounts of the code and support data structures for Jikes RVM and the benchmark itself. This impacts significantly on the cost of doing collection in the semi-space collector. This overhead could be ameliorated once Gilani [2003]'s work is incorporated into JMTk, and the Jikes RVM code can be separated into a different heap region to the application.

The performance of the Jikes RVM semi-space configuration at the maximum heap sizes, however, shows how efficient JMTk can be in its native environment, providing

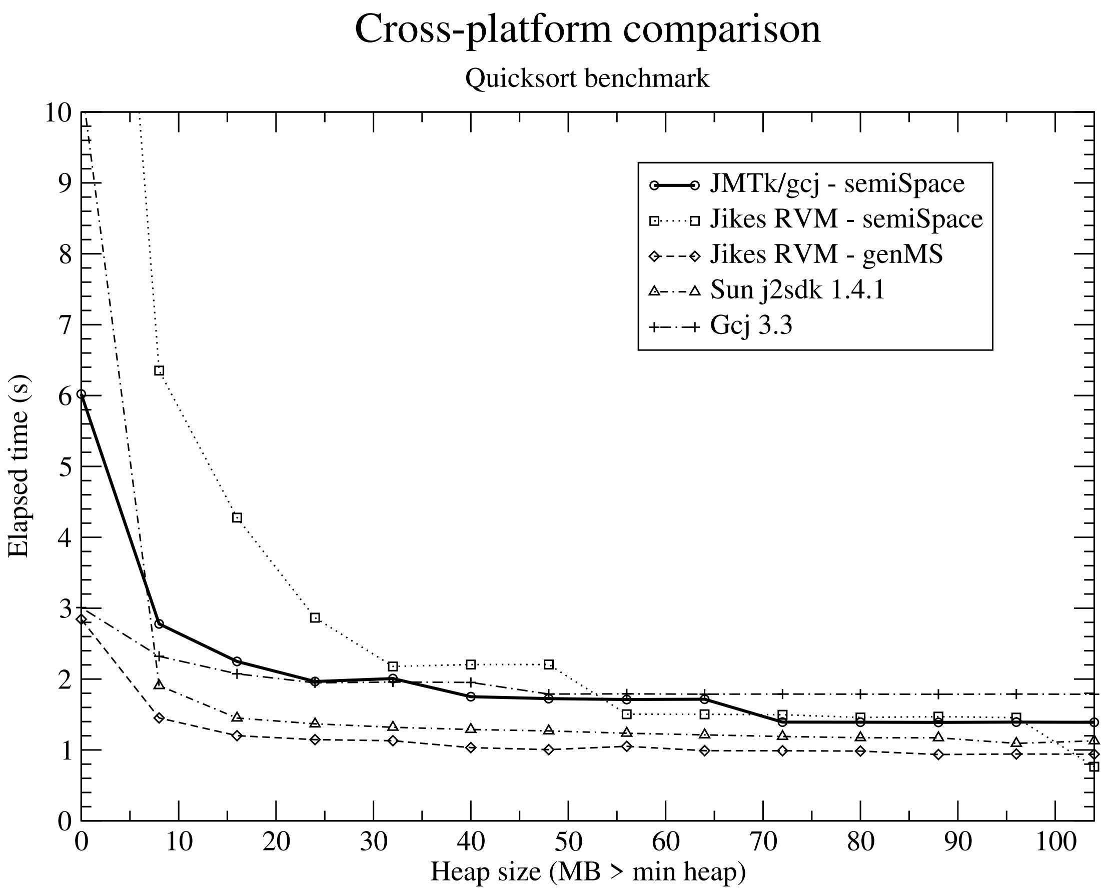


Figure 8.9: Cross platform performance comparison: the quicksort benchmark.

a 'best case' for the port. This on the other hand is not as straightforward as it sounds: the mutator code in Jikes RVM has assistance from the compiler: heap pointers can be held and passes as parameters in registers (for example) rather than stored stably in memory across all points where GC could possibly be initiated.

The Jikes RVM genMS collector shows how well JMTk collectors can perform in their native environment: this collector does not suffer as much as the semi-space collector from the overheads of Jikes RVM's Java-in-Java implementation.

The gcj comparison is quite natural, as it compares two implementations of the same algorithm with essentially the same compiler technology. The other significant comparison—between the testbed running the Boehm collector and gcj with the same collector— is shown in Appendix B. The result of this comparison shows that the testbed platform is a comparable programming environment to Java with gcj.

The Sun Hotspot JVM is probably the least justifiable comparison, but it does provide a sanity check with commercial Java virtual machine technology. Although it is consistently marginally slower than Jikes RVM's genMS collector, it is the second fastest for most heap sizes, and runs in the smallest heap size of all collectors. The collector used is a generational copying collector.

Configuration	Minimum heap (MB)	
	FTree	QSort
JMTk testbed	3	7
Jikes RVM semiSpace	15	15
Jikes RVM genMS	12	16
gcj	6	9
Sun Hotspot	1	3

Table 8.1: Minimum heap sizes for cross-platform comparisons

Despite the difficulties discussed above, the graphs of performance across machines give an interesting picture of where the portable JMTk implementation fits with respect to other systems. A valid summary might be that it is respectable, but has some room for improvement, particularly in small heaps.

8.4 The Boehm Collector

One of the major opportunities the testbed environment provides is the ability to compare the performance of the Boehm conservative collector with a variety of accurate garbage collectors. The test is far from conclusive, but some interesting observations can still be made.

The relative performance of the Boehm collector depends on heap size and object size. At the minimum heap size, it is consistently faster than all the other collectors (by up to 65%), while at larger heap sizes it is slower in benchmarks with small objects (by up to 28%), but up to 6% faster where object sizes are larger.

The validity of the comparison is weakened by several factors:

- The root set is small, and most potential roots are real roots. One of the weaknesses of conservative collectors is that they incur overhead examining false roots, and keep heap objects alive that could be collected by accurate collectors. In the JMTk testbed, the vast majority of the structures in the data segment are the actual root set. Conversely, the JMTk collectors will only trace from the roots known to be alive, while the Boehm collector will examine all of the (currently 8192) elements in the root set. The Boehm collector will also trace the stack, which is not particularly large for any of the benchmarks used, and consists mostly of low-value integers and return addresses that the Boehm collector will quickly discard as potential heap pointers.

  A more sophisticated testbed (such as would be developed if some of the future options were explored) would provide a more realistic potential root set.

- The benchmarks use a single object size. The Boehm collector uses a segregated free-list allocator, with overhead for each size class. A wider distribution of object sizes might change its performance characteristics[3].

[3] A version of the benchmarks that used a random sized 'payload' is under development but was not

These figures show the advantage of bump-pointer allocation over free-list allocation, but also that the performance overhead of the JMTk collectors is still significant. Until the JMTk mark-sweep collector has similar performance to the Boehm collector, there is still significant room for improvement.

8.5 Future work

Measuring performance is an open-ended task. The most valuable future improvements to the process focus on making benchmarks more relevant to real-life applications. There are several important approaches that can be taken.

Implementing a trace-driven benchmarking function is one possibility that may prove fruitful. While 'wall clock' performance is unlikely to be comparable to the benchmark being reproduced, direct measures of JMTk's performance (ie excluding mutator time) would be extremely useful. This would be a fairly complex system to build.

Using the profiling functions developed by Gilani [2003] to develop synthetic benchmarks that shared the same memory behaviour as a real program would also be a fruitful approach. At the least, the extent to which synthetic benchmarks replicated real program behaviour could be evaluated.

Automatic translation of C source code to the testbed idiom is another possible approach to producing more accurate benchmarks. The rules for keeping heap pointers visible to the garbage collector are systematic, and provided the program is relatively well behaved[4] this approach might be useful. Even more useful could be a transformation from another language, although this is essentially the task of writing a compiler that uses idiomatic C as an intermediate language, and would be a significant undertaking.

Simpler than the last possibility would be completing the integration of JMTk into another language runtime. The list of candidates is long, and is discussed in a later chapter.

Finally, careful examination of these results have indicated that more performance tuning is required. Several possibilities were identified in Chapter 7.

8.6 Summary

This chapter evaluated performance of four of JMTk's collectors with four synthetic benchmarks.

The Boehm conservative collector was found to be superior to the JMTk collectors in tight heaps, but between 6% better and 28% worse than the best JMTk collector in a large heap. This is not particularly disappointing considering the relative effort put into the two collectors to date.

completed in time to be used.

[4]For example, if it followed the rules required for implementing the Boehm collector.

A comparison between the same benchmarks running in different virtual machines was made, which showed that the portable JMTk collectors in the testbed runtime are comparable in performance to other Java virtual machines and garbage collectors.

Having evaluated the performance of the portable JMTk, the next chapter looks at the experience of performing a systems programming task in standard Java, while the final chapter draws some final conclusions from the work performed in this project.

Chapter 9

System programming with Java

Java, a language which emphasises type safety and machine independence, is not generally considered viable as a system programming language. Jikes RVM and similar systems succeed in performing low-level tasks because they have developed a dialect of Java that augments it with system programming constructs. This project has successfully used Java, without control over the virtual machine, and without using large scale blocks of native methods. This chapter discusses the experience of system programming in Java, its strengths and weaknesses, and features that would make life significantly easier.

9.1 System programming features of Java

The Java Language Specification [Gosling et al. 2000] presents the orthodox position: that the Java language only contains 'safe' features that can be implemented in an entirely machine independent way. For those systems that cannot be written using the Java contructs, such as device drivers, operating systems, Java virtual machines etc, Java provides native methods and a set of bindings for the C language (JNI - the Java Native Interface).

The argument that isolating unsafe code to native methods allows java to preserve its safety is difficult to sustain, as any program that calls a native method is inherently unsafe. By calling native methods to perform low-level and/or unsafe operations, as opposed to techniques employed by other languages, Java simply adds a layer of overhead and reduces the possibility of using optimization techniques such as inlining.

Ritchie [1997] describes an approach similar to that taken in this project—that as much work as possible was done in Java breaking into native code only to access memory directly, and to access hardware exceptions.

Jikes RVM, OVM and other similar projects take another approach: particular Java constructs are given direct low-level interpretations by the virtual machine, so the Java rules can be broken without breaking the Java syntax. None of this would be necessary if Java had direct support for systems-level programming.

More seriously, Java lacks several features for structured programming that one might expect a language like Java to support. There is no facility to create type synonyms for primitive types, so no matter what kind of information an `int` (for example) encodes, it must always be declared as an `int`. The Jikes RVM types `VM_Address`, `VM_Word` etc were created largely so that the ambiguity around the different meanings of integer types in the code could be removed[1].

Java also makes it difficult to express a separation of interface and implementation. While Java provides the concept of an 'interface' and an 'abstract' class, these are too restricted for JMTk's use, and have unacceptable performance penalties.

9.1.1 Other languages

Other languages that have strong typing and managed heaps, but still provide access to low level features are Modula 3 and C#.

Modula 3 [Nelson 1991] was designed specifically for system programming, and provides similar safety guarantees to Java, except in specifically marked *unsafe* modules. In an unsafe module, a programmer can:

- Allocate memory on an untraced heap. The programmer assumes responsibility for freeing it when they are finished.
- Perform unsafe type-casts using the LOOPHOLE operator, and
- Use raw pointers, with the ADDRESS type.

These features are sufficient to build programs such as operating systems [Bershad et al. 1994] in Modula 3 with only a few hundred lines of assembler. The Modula family of languages exemplifies the separation of interface and implementation.

Although not part of the language definition, the Modula 3 compilers also supports pragmas, which can be applied to various syntactic elements of a program. These provide control over various aspects of code generation.

C# provides access to similar features but at a finer grained level of control, using the 'unsafe' keyword to define 'unsafe contexts' in which pointers and type casts can be used.

9.2 JMTk's requirements

JMTk requires several features not provided by standard Java, although some are only required when JMTk is managing objects for the Java system in which it is implemented.

- Unsafe type casts. JMTk must regard the heap as a uniform set of objects, without regard for their type in other parts of the program. This is not an issue if heap objects are not Java `Objects`.
- Raw memory access. JMTk must be able to read and write from arbitrary locations in memory.

[1] Steve Blackburn, private communication.

- Unsigned arithmetic. Used for calculating with addresses, and not an inherently unsafe feature. Currently the transformed JMTk code uses an 'xor' operation on the most significant bit to scale signed integers to do an unsigned comparison.
- Atomic test-and-set instructions. Concurrent garbage collection requires more fine grained and efficient locking than can be provided by standard language synchronization techniques. Jikes RVM provides this via 'prepare' and 'attempt' methods in the `VM_Magic` class. I know of no high level language that provides this feature, except perhaps for the C 'asm' escape to assembly language.
- Memory barrier instructions. Code sections that depend on the ordering of reads and writes (implementing parallel collection algorithms) on modern out-of-order processors require the insertion of memory barriers in the code.
- Control over thread switching. Much of JMTk's code is required to be uninterruptible as it is adjusting data structures that the thread switching mechanism itself depends on. This is controlled in Jikes RVM by a pragma exception.

If Java were to provide some or all of these features, porting JMTk would have been significantly simpler. It is possible that JSR 175[2], which defines a syntax for adding annotations to Java code, will allow the current 'pragma' idiom to be done in a standard way.

9.3 Experience

In practice, using Jikes RVM's idiom approach along with the source transformation and native methods has been good, despite the limitations discussed above. None of the workarounds used have limited performance significantly (at least once the planned improvements to the source code transformer are implemented), and so far the cost of crossing language boundaries impacts mostly on code that calls 'in' to Java rather than on methods we are forced to declare **`native`**.

In conclusion, Java was found to be a realistic choice for systems programming, but would be better if a small set of features, along the lines of C# or Modula 3 were added to the language.

The next chapter concludes the thesis by summarising the contributions of this thesis, and recapping on the opportunities identified for future work.

[2] A JSR is a formal proposal for changes to the Java language.

Chapter 10

Conclusion

10.1 Contribution of this thesis

This thesis contributes to the field of memory management research by making available a portable toolkit for building garbage collectors. The transform tool used to port JMTk to the gcj environment has successfully been used by researchers at Purdue to port JMTk for use in the OpenVM Java virtual machine. Another researcher at A.N.U. is using the tools and interface to migrate JMTk to the Microsoft Rotor C# environment.

One of the obstacles to memory management research is the effort required to take an idea and realise it in a running system. A flexible object oriented component-based toolkit for building memory managers, that doesn't compromise on efficiency is a significant enabler for research. JMTk provides this for the Jikes RVM Java Virtual Machine, and the work described in this thesis will allow JMTk's advantages to be realised in other environments.

Despite the long history of research, there are still many open questions regarding the relative performance of garbage collectors. JMTk has enabled the first comprehensive study of garbage collector performance to be conducted in the Jikes RVM environment, for the Spec Jvm98 benchmark suite, and the portable version of JMTk described in this thesis will allow this comparison to be extended to other languages and environments. One frequently asked question whenever a novel technique is applied in memory management research is whether this technique is universally applicable, or whether it is only significant due to language/environment/implementation peculiarities, and these questions are not so frequently answered. Widespread use of a toolkit like JMTk would enable these issues to be settled.

Infrastructure such as JMTk tends to acquire a momentum. The success of Jikes RVM and JMTk has already attracted a significant number of researchers, and there are several additions to JMTk that are expected to be integrated in the near future. The extension of JMTk to other environments will add to this momentum, making the work of more researchers available to language implementors.

10.2 Future Work

The nature of infrastructure development like this work with JMTk is that it opens up opportunities for future research. Some of the immediately available opportunities are:

- Chapter 5 described several extensions to the source transformer that would be useful in the future, including several ways that performance could be improved.

- Chapter 7 identified several ways in which performance of the portable JMTk could be improved.

- Chapter 8 identified several ways in which benchmarking and performance measurement could be improved. Trace-based execution in particular is attractive for its debugging applications alone.

- Development of a standard interface between memory manager and language runtime, for future language and memory manager developers to adopt.

- Incorporation of JMTk into more language runtimes. The ghc and nhc98 runtime ports could be completed with a few months effort. The Rotor C# (Microsoft CLR) environment is underway.

  Of particular interest is the Melbourne University Mercury compiler. Henderson [2002] has recently developed a technique for stack scanning of gcc-generated code, and is starting to do research into memory management for logic programming languages. JMTk could significantly simplify this work.

  Any compiler built on the gcc code generator is a candidate for JMTk, by applying the Mercury solution to stack scanning. This includes SmartEiffel[1], GNU Smalltalk and the Modula-3 compilers, and the many compilers for strongly typed languages that currently use the Boehm collector.

[1] Formerly SmallEiffel.

Transform tool details

A.1 Transform specification language

```
specifications    ::= (specification ';')*
specification     ::= packageTransform
                    | importTransform
                    | methodTransform
                    | prohibition
                    | typeTransform
packageTransform  ::= 'package' identifier '=>' identifier
importTransform   ::= 'import' [identifier] '=>' [identifier]
methodTransform   ::= 'method' methodBefore '=>' methodAfter
typeTransform     ::= 'type' identifier '=>' identifier
prohibition       ::= 'prohibit' prohibitionType
prohibitionType   ::= 'import' regexp
methodBefore      ::= instDecl '.' IDENT '(' varDecls ')'
methodAfter       ::= IDENT '.' IDENT '(' varList ')'
varDecls          ::= (varDecl [',' varDecl)* ]
varDecl           ::= [identifier] IDENT
instDecl          ::= identifier | '(' varDecl ')'
varList           ::= [IDENT (',' IDENT)* ]
identifier        ::= IDENT ('.' IDENT)*
IDENT             ::= (a-zA-Z0-9_$*[])*
regexp            ::= '"' a-z|A-Z|0-9|_$*[]\. '"'
```

White space follows the standard Java rules, including single and multi-line comments.

A.2 Command line syntax

The transform tool is a command-line program with the following syntax:

```
java {java-flags} Transform {flags} spec-file {flags|parameters}
```

where `java-flags` are the standard Java virtual machine options, `spec-file` is the name of the transform specification file (which must end in `.spec`), and parameters are either the names of Java source files, or directories to process recursively. Possible flags are as follows:

`-v` Verbose output. Display the names of files as they are processed.

`-i` Incremental processing. Only process files if the source file has a modification time more recent than the corresponding output file, or the output file does not exist.

`-check` Exit with error status if a `prohibit` rule is violated.

`-o directory` Output location. When processing files from an input directory structure, they are output to a corresponding location in a directory tree with root in the most recently processed `-o` flag. There can be multiple `-o` flags, and input files are processed according to the preceding `-o` flag.

`-dt` Enables debugging output for the transformation process.

`-ds` Enables debugging output for the symbol table processing.

`-dr` Enables debugging output for the type resolution process.

`-df` Display information tracking the file lookup process.

`-showtree` Display the parsed source file using Antlr's debugging widget.

`-shownewtree` Display the transformed source file using Antlr's debugging widget.

Graphs

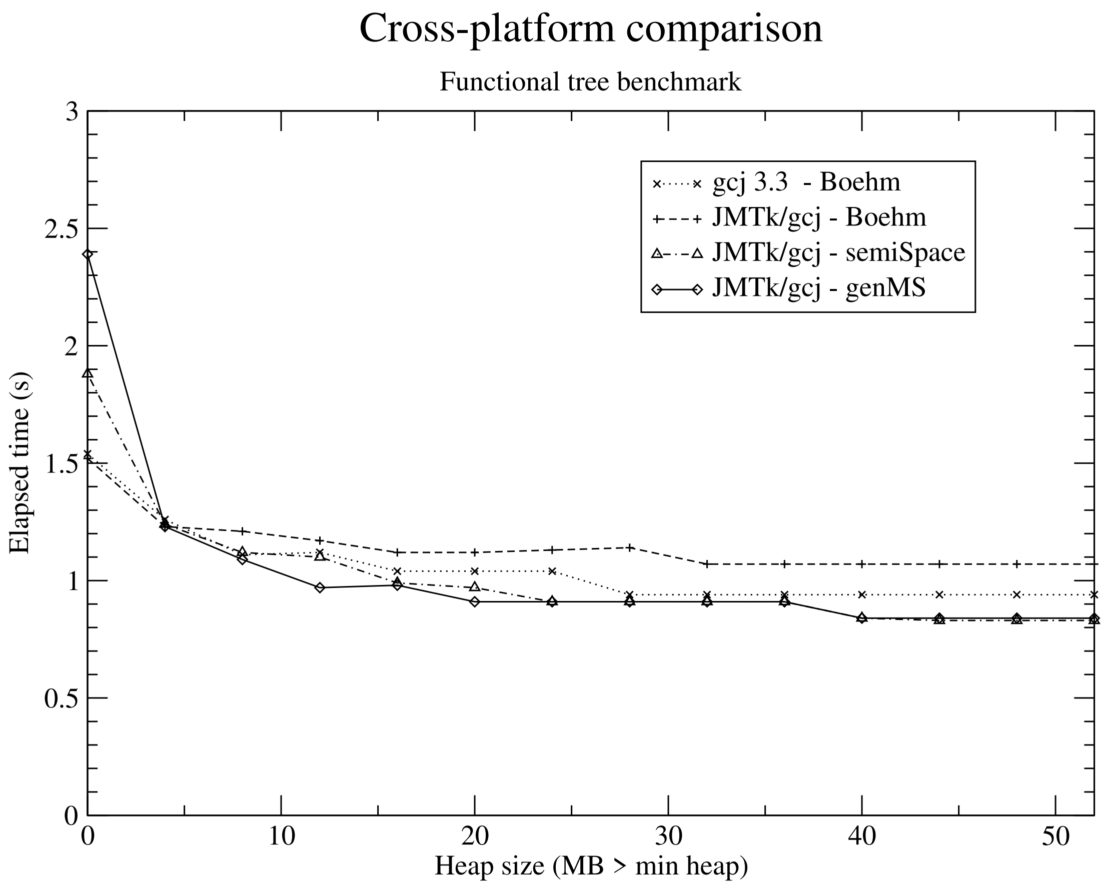


Figure B.1: gcj and Boehm comparison: FTree benchmark.

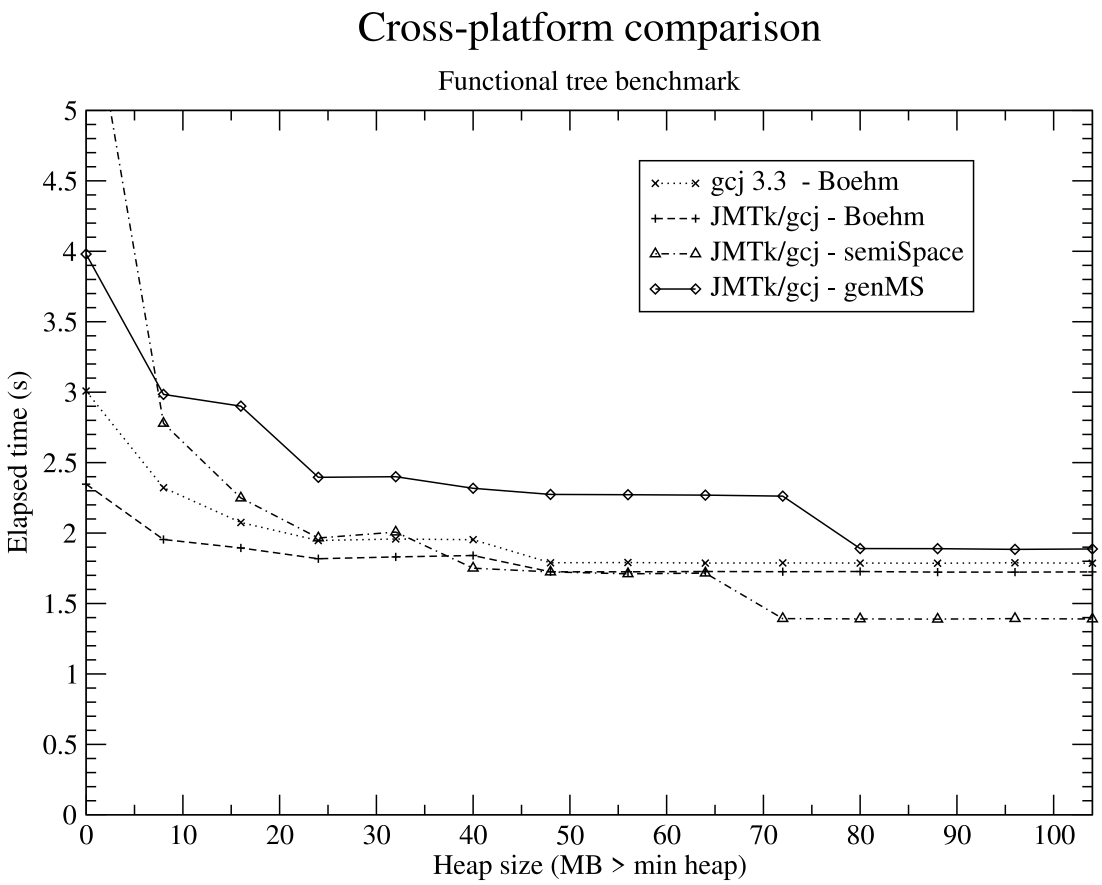


Figure B.2: gcj and Boehm comparison: QSort benchmark.

Bibliography

ALPERN, B., ATTANASIO, C. R., BARTON, J. J., COCCHI, A., HUMMEL, S. F., LIEBER, D., NGO, T., MERGEN, M. F., SHEPHERD, J. C., AND SMITH, S. 1999. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '99)* (1999), pp. 314–324. (p. 16)

ALPERN, B., ATTANASIO, C. R., BARTON, J. J., COCCHI, A., HUMMEL, S. F., LIEBER, D., NGO, T., MERGEN, M. F., SHEPHERD, J. C., AND SMITH, S. 2000. The Jalapeño virtual machine. *IBM System Journal 39*, 1 (Feb.), 211–238. (p. 16)

APPEL, A. W. 1989. Simple generational garbage collection and fast allocation. *Software—Practice and Experience 19*, 171–183. (pp. 12, 54)

ATTANASIO, C. R., BACON, D. F., COCCHI, A., AND SMITH, S. E. 2001. A comparative evaluation of parallel garbage collectors. In *Proceedings of the Fourteenth Annual Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science (June 2001). Springer-Verlag. Forthcoming. (p. 11)

AUGUSTSSON, L. 1984. A compiler for lazy ml. In *Proc. ACM Symposium on Lisp and Functional Programming* (1984). (pp. 18, 47)

BARTLETT, J. F. 1988. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC WRL. (p. 12)

BERSHAD, B. N., CHAMBERS, C., EGGERS, S. J., MAEDA, C., MCNAMEE, D., PARDYAK, P., SAVAGE, S., AND SIRER, E. G. 1994. SPIN - an extensible microkernel for application-specific operating system services. In *ACM SIGOPS European Workshop* (1994), pp. 68–71. (p. 74)

BLACKBURN, S. M., CHENG, P., AND MCKINLEY, K. S. 2004a. Myths and realities: The performance impact of garbage collection. In *Submitted for Publication to SIGMETRICS* (2004). http://www.cs.utexas.edu/users/mckinley/papers/jmtk-sigmetrics-submit-2003.ps.g z. (pp. 13, 15, 62, 67)

BLACKBURN, S. M., CHENG, P., AND MCKINLEY, K. S. 2004b. Oil and water? High performance garbage collection in Java with JMTk. In *Submitted for Publication to ICSE* (2004). http://www.cs.utexas.edu/users/mckinley/papers/jmtk-icse-submit-2003.ps.gz. (pp. 13, 15)

BLACKBURN, S. M., JONES, R., MCKINLEY, K. S., AND MOSS, J. E. B. 2002. Beltway: Getting around garbage collection gridlock. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Berlin, Germany, June 2002). ACM. (pp. 11, 13)

BLACKBURN, S. M. AND MCKINLEY, K. S. 2003. Ulterior Reference Counting: Fast Garbage Collection without a long wait (Oct. 2003). OOPSLA 2003. (pp. 11, 16)

BOEHM, H.-J. AND WEISER, M. 1988. Garbage collection in an uncooperative environment. *Software Practice and Experience 18*, 9, 807–820. (p. 11)

CHENEY, C. J. 1970. A non-recursive list compacting algorithm. *Communications of the ACM 13*, 11 (Nov.), 677–678. (p. 9)

CLINGER, W. D. AND HANSEN, L. T. 1997. Generational garbage collection and the radioactive decay model. In *SIGPLAN Conference on Programming Language Design and Implementation* (1997), pp. 97–108. (p. 11)

CLINGER, W. D. AND HANSEN, L. T. 2002. An experimental study of renewal-older-first garbage collection. In *Proceedings of the International Conference on Functional Programming* (Oct. 2002). ACM. (pp. 11, 12)

COLLINS, G. E. 1960. A method for overlapping and erasure of lists. *Communications of the ACM 3*, 12 (Dec.), 655–657. (p. 8)

DEUTSCH, L. P. AND BOBROW, D. G. 1976. An efficient incremental automatic garbage collector. *Communications of the ACM 19*, 7 (July), 522–526. (p. 9)

DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M. 1978. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM 21*, 11 (Nov.), 965–975. (p. 7)

FENICHEL, R. R. AND YOCHELSON, J. C. 1969. A lisp garbage collector for virtual memory systems. *Communications of the ACM 12*, 11 (Nov.), 611–612. (p. 9)

FLACK, C., HOSKING, T., AND VITEK, J. 2003. Idioms in ovm. Technical Report CSD-TR-03-017, Department of Computer Sciences, Purdue University. (p. 17)

GARRIDO, A. AND JOHNSON, R. 2003. Refactoring C with conditional compilation. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003)* (Montreal, Canada, Oct. 2003), pp. 323–326. IEEE. (p. 43)

GILANI, F. 2003. Title unknown. Master's thesis, Australian National University. (pp. 68, 71)

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2000. *The Java Language Specification*. Addison-Wesley. (p. 73)

HANSEN, W. J. 1969. Compact list representation: Definition, garbage collection and system implementation. *Communications of the ACM 12*, 9 (Sept.), 499–507. (p. 9)

HENDERSON, F. 2002. Accurate garbage collection in an uncooperative environment. In *Proceedings of the third international symposium on Memory management* (2002), pp. 150–156. ACM Press. (p. 78)

HUDSON, R., MOSS, E., DIWAN, A., AND WEIGHT, C. F. 1991. A language-independent garbage collector toolkit. Technical Report TR 91-47, University of Massachusetts at Amherst. (p. 13)

JOHNSSON, T. 1984. Efficient compilation of lazy evaluation. In *Proc. ACM SIGPLAN '84 Symposium on Compiler Construction* (1984). SIGPLAN Notices **19**(6). (pp. 18, 47)

JONES, R. AND LINS, R. 1996. *Garbage Collection*. John Wiley and Sons. (pp. 5, 8, 12)

LEVANONI, Y. AND PETRANK, E. 2001. An on-the-fly reference counting garbage collector for java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01).* (2001), pp. 367–380. (p. 9)

LIEBERMAN, H. AND HEWITT, C. 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM 26*, 6, 419–429. (p. 10)

MCCARTHY, J. 1960. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM 3*, 4 (April), 173–197. (pp. 7, 8)

NELSON, G. Ed. 1991. *Systems Programming with Modula-3*. Prentice Hall. (p. 74)

PARR, T. 2002. ANTLR: Another Tool for Language Recognition. `http://www.antlr.org/`. (p. 41)

PARR, T. J. 1994. An overview of SORCERER: A simple tree-parser generator. Technical report, `http://www.antlr.org/`. (p. 41)

PARR, T. J. AND QUONG, R. W. 1995. Antlr: A Predicated-LL(k) Parser. *Software—Practice And Experience 25*, 7 (July), 789–810. (p. 41)

PEYTON JONES, S. L. 1987. *The implementation of Functional Programming Languages*. Prentice Hall. (pp. 18, 47)

PEYTON JONES, S. L. 1992. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of Functional Programming 2*, 127–202. (p. 47)

PEYTON JONES, S. L., MARLOW, S., AND REID, A. 1999. The STG runtime. http://www.haskell.org/ghc/. (p. 48)

QUINANE, L. 2003. Honours thesis. A.N.U. (p. 9)

RITCHIE, S. 1997. Systems Programming in Java. *IEEE Micro 17*, 3 (May/June), 30–35. (pp. 16, 73)

ROVNER, P. 1985. On adding garbage collection and runtime types to a strongly typed, statically-checked, concurrent language. Technical report CSL-84-7 (July), Xerox PARC, Palo Alto, CA. (p. 5)

SANSOM, P. 1991. Dual-mode garbage collection. In H. GLASER AND P. H. HARTEL Eds., *Proceedings of the Workshop on the Parallel Implementation of Functional Languages* (Southampton, UK, 1991), pp. 283–310. Department of Electronics and Computer Science, University of Southampton. (p. 12)

STEFANOVIĆ, D. 1993. The garbage collection toolkit as an experimentation tool. OOPSLA 1993 Workshop on Memory Management and Garbage Collection. (p. 13)

STEFANOVIC, D., MCKINLEY, K. S., AND MOSS, J. E. B. 1999. Age-based garbage collection. In *OOPSLA '99* (1999), pp. 370–381. (p. 11)

STEFANOVIĆ, D., MOSS, J. E. B., AND MCKINLEY, K. S. 1998. Oldest-first garbage collection. Technical Report 98–81, University of Massachusetts. (p. 11)

UNGAR, D. M. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices 19*, 5 (April), 157–167. (p. 10)

WADSWORTH, C. P. 1971. *Semantics and Pragmatics of the lambda calculus*. PhD thesis, Oxford University. (p. 48)

WARRINGTON, I. 2003. Honours thesis. A.N.U. (p. 9)

WILSON, P. R. 1992. Uniprocessor garbage collection techniques. In H. BAKER Ed., *Proceedings of the International Workshop on Memory Management*, Volume 637 of *Lecture Notes in Computer Science* (St Malo, France, Sept. 1992). Springer-Verlag. (p. 86)

WILSON, P. R. 1994. Uniprocessor garbage collection techniques. Expanded version of [Wilson 1992]. (p. 12)

WILSON, P. R., JOHNSTONE, M. S., NEELY, M., AND BOLES, D. 1995. Dynamic storage allocation: A survey and critical review. In H. BAKER Ed., *Proceedings of International WOrkshop on Memory Management, IWMM'95, Kinross, Scotland*, Volume 986 of *Lecture Notes in Computer Science* (Sept. 1995). Springer-Verlag. (p. 6)

ZORN, B. 1991. The effect of garbage collection on cache performance. Technical Report CU–CS–528–91, Department of Computer Science, Boulder, Colorado. (p. 5)

Never index your own book

She said that indexing was a thing that only the most amateurish author undertook to do for his own book. I asked her what she thought of Philip Castle's job.

'Flattering to the author, insulting to the reader,' she said. 'In a hyphenated word,' she observed, with the shrewd amiability of an expert, ' "*self-indulgent*". I'm always embarrassed when I see an index an author has made of his own work.'

'Embarrassed ?'

'It's a revealing thing, an author's index of his own work,' she informed me. 'It's a shameless exhibition—to the *trained* eye.'

'She can read character from an index,' said her husband.

'Oh ?' I said. 'What can you tell about Philip Castle ?'

She smiled faintly. 'Things I'd better not tell strangers.'

Kurt Vonnegut
Cat's Cradle