The Design and Construction of High Performance Garbage Collectors

Robin Garner

May 2012

A thesis submitted for the degree of
Doctor of Philosophy at
The Australian National University

Australian
National
University

Typeset in Palatino by TEX and LATEX 2ε.

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

Robin Garner
13 July 2011

Abstract

Garbage collection is a performance-critical component of modern language implementations. The performance of a garbage collector depends in part on major algorithmic decisions, but also significantly on implementation details and techniques which are often incidental in the literature.

In this dissertation I look in detail at the performance characteristics of garbage collection on modern architectures. My thesis is that a thorough understanding of the characteristics of the heap to be collected, coupled with measured performance of various design alternatives on a range of modern architectures provides insights that can be used to improve the performance of any garbage collection algorithm.

The key contributions of this work are: 1) A new analysis technique (replay collection) for measuring the performance of garbage collection algorithms; 2) a novel technique for applying software prefetch to non-moving garbage collectors that achieves significant performance gains; and 3) a comprehensive analysis of object scanning techniques, cataloguing and comparing the performance of the known methods, and leading to a new technique that optimizes performance without significant cost to the runtime environment.

These contributions are applicable to a wide range of garbage collectors, and can provide significant measurable speedups to a design point where each implementer in the past has had to trust intuition or their own benchmarking. The methodologies and implementation techniques contributed in this dissertation have the potential to make a significant improvement to the performance of every garbage collector.

Contents

Abstract **v**

1 Introduction **1**
1.1 Problem Statement . . . 1
1.2 Scope and Contributions . . . 3
1.2.1 Workload Evaluation . . . 3
1.2.2 Software Prefetch for Garbage Collection . . . 3
1.2.3 Object Scanning Techniques . . . 3
1.3 Thesis Outline . . . 4

2 Background **5**
2.1 Garbage Collection . . . 5
2.1.1 Terminology . . . 7
2.1.2 Memory allocation . . . 8
2.1.2.1 Free-list allocators . . . 8
2.1.2.2 Monotonic allocators . . . 9
2.1.2.3 Multi-threaded allocators . . . 9
2.1.3 Canonical garbage collection algorithms . . . 10
2.1.3.1 Mark-sweep . . . 10
2.1.3.2 Reference counting . . . 11
2.1.3.3 Evacuating . . . 12
2.1.3.4 Compacting . . . 13
2.1.3.5 Mark-region . . . 14
2.1.3.6 Generational . . . 14
2.1.4 Object Scanning . . . 15
2.2 Infrastructure . . . 16
2.2.1 Jikes RVM . . . 16
2.2.1.1 Low-level Programming . . . 17
2.2.1.2 Replay Compilation . . . 17
2.2.2 MMTk . . . 18
2.2.2.1 The MMTk Free-list Allocator . . . 18
2.2.2.2 Credibility of MMTk As An Experimental Platform . . . 19
2.3 Evaluation Methodology . . . 19
2.3.1 Benchmarks . . . 20
2.3.1.1 The SPEC Benchmarks . . . 20
2.3.1.2 The DaCapo Benchmarks . . . 21
2.3.2 Evaluation Methodology . . . 21

2.4 Summary . . . 22

3 Garbage Collector Performance 23
3.1 Introduction . . . 23
3.2 The Replay Tracing Framework . . . 24
3.3 Results . . . 25
3.3.1 The Composition of the Tracing Loop . . . 25
3.3.1.1 Replay Scenarios . . . 25
3.3.1.2 Tracing Costs . . . 28
Framework Overhead . . . 28
Experiments . . . 28
3.3.1.3 Results . . . 29
3.3.2 Mark State Implementations . . . 31
3.3.3 Heap Traversal Order . . . 33
3.3.3.1 Experiments . . . 33
3.3.3.2 Results . . . 35
Depth-first versus breadth-first . . . 35
FIFO buffer . . . 36
Work Packet . . . 36
Scan Direction . . . 36
Edge Enqueuing . . . 37
3.3.3.3 Conclusion . . . 39
3.4 Summary . . . 39

4 Effective Software Prefetch 41
4.1 Introduction . . . 41
4.2 Related work . . . 42
4.3 Key Mark-Sweep Design Choices . . . 44
4.3.1 Allocation . . . 44
4.3.2 Mark state . . . 44
4.3.3 Sweep . . . 44
4.3.3.1 Block marks . . . 45
4.3.3.2 Cyclic mark state . . . 45
4.3.4 Work queue . . . 46
4.4 Software Prefetching . . . 46
4.4.1 Prefetching For GC Tracing . . . 47
4.5 Edge Order Traversal . . . 47
4.6 Performance Results . . . 48
4.7 Robustness: Experiences With Other Code Bases . . . 52
4.8 Summary . . . 52

5 Object Scanning 53
5.1 Introduction . . . 54
5.2 Related Work . . . 55

5.3 Analysis of Scanning Patterns . . . 56
5.3.1 Analysis Methodology . . . 57
5.3.2 Encoding and Counting Patterns . . . 57
5.3.3 Benchmarks . . . 60
5.3.4 Reference Pattern Distributions . . . 60
5.3.5 Reference Field Count Distributions . . . 63
5.4 Design Alternatives . . . 63
5.4.1 The Jikes RVM Scanning Mechanism . . . 63
5.4.2 Inlining Common Cases . . . 64
5.4.3 Compiled vs. Interpreted Evaluation . . . 65
5.4.4 Encoding and Packing of Metadata . . . 65
5.4.5 Indirection to Metadata . . . 66
5.4.6 Object Layout Optimizations . . . 66
5.5 Methodology . . . 67
5.5.1 Hardware Platforms . . . 68
5.5.2 Configurations . . . 68
5.6 Results . . . 69
5.6.1 Inlining Common Cases . . . 70
5.6.2 Compiled vs. Interpreted Evaluation . . . 70
5.6.3 Encoding and Packing of Metadata . . . 71
5.6.4 Indirection to Metadata . . . 71
5.6.5 Object Layout Optimizations . . . 72
5.6.6 Conclusion . . . 72
5.7 Summary . . . 75

6 Conclusion **79**
6.1 Future Work . . . 79
6.1.1 Prefetch . . . 80
6.1.2 Scanning . . . 80

Bibliography **81**

List of Figures

1.1 Architectural dependence on GC performance . . . 2

2.1 Mark-Sweep Garbage Collection . . . 11

3.1 The Standard Tracing Loop . . . 26
3.2 Replay Scenarios . . . 27
3.3 Replay Scenario for Evaluating Mark-state Implementations . . . 32
3.4 The *Edge-Enqueuing* Tracing Loop . . . 35
3.5 Effects of traversal order: major design choices . . . 36
3.6 Effects of a FIFO buffer . . . 36
3.7 Effects of partial breadth-first (work packet) traversal order . . . 37
3.8 Reversing the order of scanning of fields . . . 37
3.9 Edge enqueuing. . . . 37
3.10 Edge enqueuing with a FIFO buffer . . . 38
3.11 Edge enqueuing. Effects of partial breadth-first order . . . 38

4.1 The FIFO-Buffered Prefetch Queue [Cher et al., 2004]. . . . 43
4.2 Comparing the Standard and Edge Enqueuing Tracing Loops . . . 48
4.3 Normalized GC time vs. prefetch distance . . . 49
4.4 Normalized total time vs. prefetch distance . . . 50
4.5 Relative total execution time as a function of heap size . . . 51

5.1 Cumulative frequency distribution curves for *reference layout patterns* . . . 59
5.2 Cumulative frequency distribution curves for reference *field counts* . . . 62
5.3 Objects and Per-Class Metadata Structure in Jikes RVM. . . . 64
5.4 The default scanning loop in Jikes RVM. . . . 64
5.5 Unoptimized and optimized versions of scanning code. . . . 65
5.6 Inlining common cases . . . 70
5.7 Specialization . . . 70
5.8 Header encodings . . . 71
5.9 Levels of indirection . . . 72
5.10 Object layout optimizations . . . 73
5.11 Six well-performing designs . . . 74
5.12 Six well-performing designs—per-benchmark, Core i5 and Core 2 processors . . . 76
5.13 Six well-performing designs—per-benchmark, Phenom and Atom processors . . . 77

List of Tables

2.1 Comparative GC performance, MMTk (Jikes RVM) vs. Boehm (GCJ). GC Throughput in MB/s 19

3.1 Tracing loop time as a percentage of total GC time. 24
3.2 Hardware Platforms used for the Replay Tracing Loop Costs Experiments 28
3.3 Elapsed Time for Various Scenarios for Two Design Points, Normalized to the Synchronized Mark Scenario. 29
3.4 Costs for Two Designs, Showing Time, Retired Instructions, L1 & L2 Misses, Normalized to Each Mark Scenario. 30
3.5 Elapsed Time for Various Scenarios for Four Design Points, Normalized to the Synchronized Mark Scenario. 32
3.6 Cost of The Mark Mechanism Alone for Four Design Points, Each Normalized to Cost of Entire Mark Scenario. 33
3.7 Hardware platforms used for heap traversal order experiments. 35

4.1 Hardware Platforms for prefetch experiments 49

5.1 Detailed *reference layout pattern* distributions for 'references first' object layout 58
5.2 Detailed reference *field count* distributions 61
5.3 Hardware platforms for scanning experiments. 68
5.4 Object scanning configurations evaluated 68
5.5 Header encoding: Percentage of objects covered by the schemes evaluated. 69

Chapter 1

Introduction

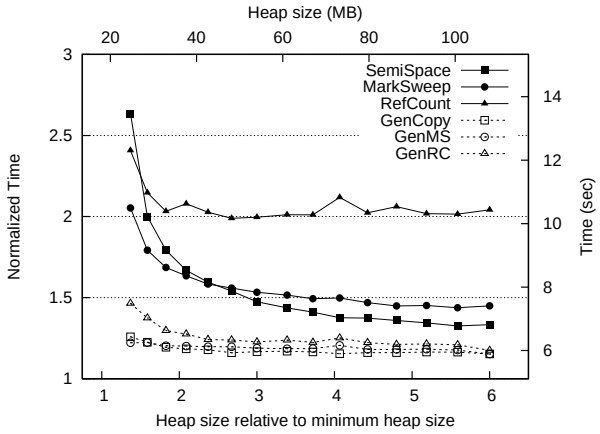
This thesis addresses the problem of designing high performance implementations of garbage collection algorithms on modern hardware.

1.1 Problem Statement

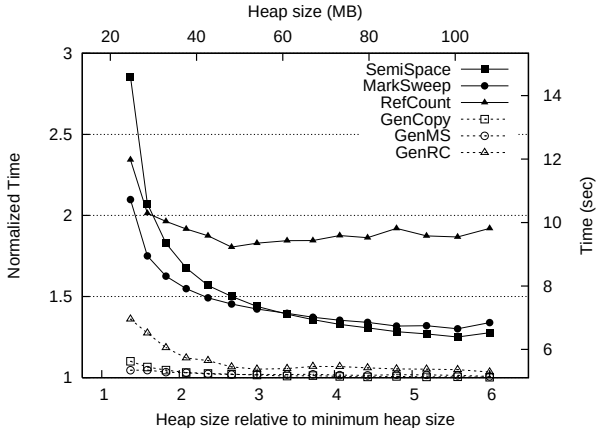
Garbage collection is an increasingly important feature of modern programming languages. Since its inception in LISP [McCarthy, 1960], there has been a huge volume of research into garbage collection. Most of this research has focused on the algorithm used, rather than the details of its implementation.

The performance characteristics of garbage collectors depend strongly on processor architecture, particularly on the speed ratio between Register/ALU operations and memory operations at the various levels of cache. For example, see Figure 1.1, reproduced from Blackburn et al. [2004a], which shows the performance of six garbage collectors on three architectures across a range of heap sizes. The data point of interest is the point where the curves for MS (Mark-Sweep) and SS (Semi-Space) intersect. Mark-sweep is a high throughput collector with poor mutator locality, while semi-space is a low throughput collector with excellent mutator locality. In a large heap, the garbage collector runs less frequently, and mutator performance dominates overall performance. As the heap size decreases the garbage collector performance comes to dominate. On architectures where locality is more important, the crossover point will occur at a smaller heap size, and the semi-space collector will outperform mark-sweep over a larger range of heap sizes. On an architecture where locality was unimportant (e.g. on historical machines with a flat memory hierarchy) mark-sweep would perform better than semi-space across all heap sizes. This architectural dependence means that the measurement and analysis that informs garbage collector design decisions is as important as overall performance, if not more so—because while a particular design may be superseded by advances in technology, the analysis techniques that led to the design are likely to still be relevant.

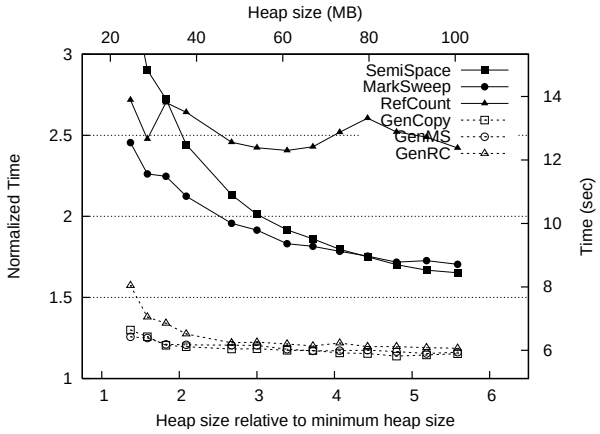
Another important factor in garbage collector performance is the workload, i.e. the composition of the heap in its target applications. For research in the field, this effectively means the benchmarks used to evaluate the work. Garbage collector implementations that look carefully at the heap composition can be more effectively optimized



(a) Pentium 4, 2.6GHz, 8KB L1, 256KB L2



(b) AMD Athlon 2600+, 1.9GHz, 64KB L1, 512KB L2



(c) Power PC 970, 1.6GHz, 32KB L1, 512KB L2

Figure 1.1: Architectural dependence on GC performance. Total running time on three architectures, geometric mean of the Spec JVM 98 benchmarks. Reproduced from Blackburn et al. [2004a].

to make the frequent case fast, and achieve speedups that would not otherwise be possible.

1.2 Scope and Contributions

The aim of my research has been to identify ways in which the performance of garbage collection can be improved, and to measure and quantify the improvement. To do this, I have used the Jikes RVM Java virtual machine, because this is a high-performance, freely available implementation of one of the most relevant modern programming languages. While the specific quantitative results may change, the methodologies and results should be applicable to other Java implementations, and to other programming languages.

1.2.1 Workload Evaluation

One of my key contentions is that the design of high performance garbage collectors must be driven by a thorough empirical understanding of the workload and the performance of the system being implemented. During the course of my PhD I have been deeply engaged in the development of the DaCapo benchmark suite [Blackburn et al., 2006, 2008].

This dissertation introduces the technique of *replay collection* as a methodology for evaluating the performance of a garbage collector. This technique is used to examine several aspects of the performance of a collector, and results from this evaluation inform the implementation techniques presented in later chapters.

1.2.2 Software Prefetch for Garbage Collection

Several authors have applied software prefetch to garbage collection, and have achieved modest speedups at best. We show how a little-used algorithm for heap traversal, coupled with buffered prefetch and a new object metadata implementation combine to yield significant, consistent speedups on a large class of garbage collectors on a wide range of benchmarks and hardware platforms.

1.2.3 Object Scanning Techniques

We comprehensively investigate the second most significant component of the garbage collection loop, reference field identification or object scanning. The evaluation and comparison of all published object scanning methods is a significant contribution to the garbage collection literature. We also contribute a new technique—using a small field in the object header to encode frequent object patterns—which coupled with scanning method specialization gives significant, consistent speedups over standard approaches.

1.3 Thesis Outline

The body of this thesis is structured around the three contributions outlined above.

Chapter 2 provides an overview of garbage collection with particular focus on the techniques discussed in this thesis. It also provides background on the benchmarks used for memory management research, and particularly the DaCapo benchmarks project. Further, this chapter describes Jikes RVM and MMTk, and outlines briefly the evaluation techniques used in this research.

Chapters 3–5 comprise the main body of the thesis. They cover the three major research contributions outlined in Section 1.2.

Finally, Chapter 6 concludes the thesis, summarizing how my contributions have addressed the problem of designing and implementing high performance garbage collectors, and identifying future research directions.

Chapter 2

Background

It will take more time:
If you touch it, it will break...
Software is wily.

Rose [2008]

This chapter provides the background material necessary to place the research contributions in context, and to describe the specific research infrastructure used in the remainder of the thesis.

Section 2.1 begins with a brief introduction to the field of garbage collection, with emphasis on the specific techniques that form the basis of this thesis. Section 2.2 describes Jikes RVM, the research Java virtual machine in which this work was implemented, and MMTk, the memory manager used in Jikes RVM. Section 2.3 describes the benchmarks and experimental methodology used to evaluate the designs presented in later chapters.

2.1 Garbage Collection

Automatic memory management, or *garbage collection*, has been a feature of many programming languages since early versions of LISP [McCarthy, 1960], and is now in widespread use due to the popularity of languages such as Java, C#, Python and Perl. This section provides a brief overview of garbage collection terminology, algorithms, and mechanisms. For a more complete discussion of the fundamentals of garbage collection see Jones and Lins [1996]; Jones et al. [2011]; Wilson [1992].

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 in terms of several attributes, including:

- 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 stop 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 (precise collectors) or can run with limited or no information (conservative and ambiguous roots collectors).
- Whether at every invocation 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 pointer mutation rate.

There are some drawbacks to automatic memory management. Manual memory management potentially uses less memory, since objects can be freed as soon as they are no longer used, rather than waiting until many objects are ready to be reclaimed, as most garbage collectors do. In practice automatic memory management introduces a space-time tradeoff, where a larger heap allows for less frequent garbage collection and consequently better performance. Manual memory management is, however, not without its own space overheads. In a complex system it can often be very difficult to determine when an object is no longer in use.

One system for manual memory management that helps the programmer track objects that are in use is *talloc* [Tridgell, 2004], which solves the problem by adding an additional 32-byte header (60 bytes in a 64-bit system) to objects, over and above the header used by the underlying 'malloc' allocator, and reclaims unused objects in a tree structure. Allocating memory in talloc requires specifying a parent *talloc context*, and when a talloc object is freed, all the objects allocated within its context are also freed. Using talloc correctly requires that all functions that allocate memory be replaced with their talloc equivalents, and that code that allocates memory that lives beyond its enclosing context needs to export it to the external context. In exchange for this additional complexity, talloc provides considerable debugging assistance in tracking down memory leaks and use-after-free errors.

Hertz and Berger [2005] explore the tradeoff between automatic and manual memory management in an artificial setting, using a memory trace from a previous run of the benchmark to determine when objects become unreachable and inserting explicit calls to free objects at the moment they become unreachable. This analysis, while interesting in itself, ignores the enormous programming effort and overhead (as in talloc above) required to track the ownership of objects and determine when they are no longer used. The most frequently cited study is Rovner [1985], who found that Mesa programmers spent 40% of their time implementing memory management procedures and finding errors related to explicit storage management.

Another often cited problem with garbage collected systems is a lack of deterministic finalization. Finalization is a feature of memory management systems that allows

a programmer to provide code that the system undertakes to execute after an object is found to be unreachable but before it is reclaimed. In an explicit memory management system, finalizers are executed at the moment the programmer frees the object, while in a garbage collected system this only occurs sometime after the last reference to the object is removed. In some memory managers the finalizer may only be called when the program exits, since memory is generally only freed in response to resource exhaustion. In my opinion this claimed drawback is rather specious, since while the finalization feature (and its slightly more usable weak reference feature) have limitations, there is nothing stopping the programmer from executing finalization code at the point they determine that the object is about to become unreachable. The advantage that garbage collection supported finalization adds is that a finalizer will eventually be run, even when an object becomes unreachable without the programmer noticing and arranging for manual finalization.

As with finalization, another potential drawback of garbage collection is that it can provide the programmer with a false sense of security, leaving them the illusion that memory management is purely a runtime-system issue. In practice, garbage collected systems can have memory leaks if the programmer maintains references to memory that is no longer in use. This can sometimes be subtle, such as when the compiler's liveness analysis is less than perfect—a variable that is no longer in scope may maintain a pointer to a large data structure. These memory leaks are easily fixed by explicitly breaking the link to the data structure at the point the programmer no longer needs it, and the garbage collector still frees the programmer from leaks due to genuinely unreachable memory.

2.1.1 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 generally 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 by McCarthy [1960]. Following Dijkstra et al. [1976], from the point of view of the garbage collector, the term *mutator* refers to the application, 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 length of time that the garbage collector interrupts the main program. Collectors that must stop the mutator to perform collection work are known as *stop the world* collectors, as compared to *concurrent* or *on-the-fly* collectors. Of particular interest to real-time systems is the *maximum pause time*, because when this can be reduced (or preferably bounded) the system will be more responsive. The term *GC time* is used to denote the time when the garbage collector is running.

A garbage collector that always checks the liveness of all objects in the heap is known as a *full heap* collector, as compared to a *generational* or *incremental* collector.

Some garbage collectors require interaction with the running mutator. These interactions are known generically as *barriers*. The most common form of barrier is a

write barrier, which is invoked whenever the mutator writes to a field in an object in the heap. Less commonly used are *read barriers*, invoked on a read of a field, and *stack barriers* invoked on return from a method call. Barriers are typically only required on reference fields, although some collectors require barriers on non-reference fields.

2.1.2 Memory allocation

Memory allocation is in itself an extensive field, and the standard reference is the survey paper by Wilson et al. [1995]. The key issues for memory allocation are locality, 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.

When discussing memory allocation, one of the key efficiency issues is *fragmentation*, which refers to memory wasted by the allocator. There are two types of fragmentation: *internal fragmentation* occurs when the allocator returns a larger region of memory than the user requested, resulting in part of the allocated cell being unused; *external fragmentation* occurs when free memory is available in the spaces between allocated regions, but which is too small to satisfy an allocation request.

2.1.2.1 Free-list allocators

In systems where all memory cells are the same size (such as LISP), a bitmap can be used to indicate which cell of memory is free. Object allocation in this style of allocator searches the bitmap for the first free bit, then returns the address of the corresponding block. The most widely used allocation strategy in explicitly managed heaps is the free-list allocator, which chains together free blocks of memory into a list. The design space for free-list allocators is large, and involves complex trade-offs between space and time efficiency.

At one end of the design spectrum for free-list allocators is an *exact first fit* algorithm. Initially the allocator assigns the entire heap to a single large contiguous block, and satisfies allocation requests by returning an initial segment of the unused memory. As the program returns objects to the memory manager, it chains them on a list, in ascending order of size. When the allocator processes a new request, it scans the free list, and returns the first block that is the same or greater size than the request. If the block found is larger than the request, the allocator splits the block, and returns the unused portion to the free list. This algorithm suffers badly from external fragmentation [Robson, 1975], and coalescing of small unused chunks into larger fragments, while expensive to implement, is necessary to make this practical.

At the other end of the spectrum is a segregated fits algorithm, that rounds up memory requests to a series of thresholds. This class of algorithm has very good allocation and de-allocation performance, but can suffer from internal fragmentation if cell sizes are a poor match for application allocation request patterns. This approach is typified by the Kingsley allocator [Wilson et al., 1995], an implementation of the

buddy system [Knowlton, 1965], a power of 2 segregated fits allocator with extremely fast performance, but which has a worst case internal fragmentation penalty of 50% of available memory. The Kingsley allocator was the default implementation of `malloc` in BSD Unix 4.2.

The best known free-list allocators, such as the Lea allocator used by GNU libc [Lea, 1998], 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 128KB, Lea uses the underlying operating systems virtual memory allocation functions.

One disadvantage of free-list allocators is that objects allocated contemporaneously are not guaranteed to be collocated in memory. That is, temporal locality does not lead to spatial locality. Objects allocated close in time are frequently accessed together during execution, so free-list allocators lead to poor cache hit rates, and hence poor mutator performance compared to monotonic allocators [Blackburn et al., 2004a].

2.1.2.2 Monotonic allocators

Much faster than a free-list allocator is a *monotonic* or *bump-pointer* allocator. The allocator acquires a region of unused memory, maintains a single pointer to the lowest free address in the region, and after allocating an object, increments ('bumps') it by the size of the object just allocated. When memory is exhausted, a *garbage collection* mechanism (such as the semi-space collector described below in Section 2.1.3.3) is invoked, which frees a large contiguous region or regions so that allocation can resume. In a practical collector, the bump-pointer will allocate regions at some chunk size, e.g. 32KB in MMTk. A larger chunk size minimizes chunk allocation overhead and multi-threading overhead, while a smaller chunk size minimizes wasted space, especially important in multi-threaded allocators. Bump-pointer allocation is significantly faster than free-list allocation, and has better mutator performance [Blackburn et al., 2004a].

2.1.2.3 Multi-threaded allocators

Multi-threaded allocators raise several issues not relevant to single-threaded allocators [e.g. Alpern et al., 2000; Berger et al., 2000; Garthwaite and White, 1998]. First, sharing a single free-list pointer or bump-pointer in a multi-threaded system incurs significant synchronization overhead. Second, in the majority of applications, most objects allocated by a thread are used only by that thread, so interleaving objects allocated by different threads can lead to false sharing of cache lines and consequent poor performance even when no objects are shared between threads.

Most multi-threaded systems use some version of *private allocators*, where each thread allocates a moderate size chunk of memory from a global pool using synchronization, and then allocates objects from that chunk without synchronization. Because these chunks are typically much larger than a cache line, false sharing can only occur where objects are actually shared by multiple threads. Synchronization overhead is

greatly reduced provided that the thread-local chunks are sufficiently large, although there is an evident space/time trade-off.

2.1.3 Canonical garbage collection algorithms

This section describes the basic algorithms used to construct garbage collectors.

Dijkstra et al. [1976] introduced the *tri-colour abstraction*, which is useful in discussing and comparing garbage collection algorithms. The abstraction defines objects in the heap to be one of three colours, traditionally white, grey and black. A classic full-heap stop-the-world collector begins with all objects coloured white. The collector iterates through the *root set*, i.e. the set of references into the collected heap—static program variables, stack-allocated local variables and registers, colouring all referenced objects grey. The collector then iterates, choosing an object in the grey set and colouring all of its white referents grey, then colouring the chosen object black. The process terminates when the grey set is empty. This forms the transitive closure of the root set over the heap, identifying all objects which are reachable from variables in the program. At this point, any object still coloured white is unreachable and can be reclaimed. Collectors differ largely in how these sets are represented, and what happens when an object is moved from one set to another.

We use the term *tracing* to refer to this transitive closure process. A *tracing collector* is one that determines liveness purely via a transitive closure process, as opposed to a reference counting collector (see Section 2.1.3.2 below).

2.1.3.1 Mark-sweep

The first garbage collection algorithm was created as part of the LISP system [McCarthy, 1960], and is today known as mark-sweep. The mark-sweep algorithm associates a single bit flag called the *mark bit* with each object, initially set to zero, corresponding to white in the tricolour abstraction. The initial scan of the root set proceeds by setting the referenced object's mark bit to one and adding it to the work list. The grey objects in the abstraction correspond to the contents of the work list. During the closure phase (known in a mark-sweep collector as the *mark phase*), the collector iterates until the work list becomes empty. In each iteration an object is removed from the work list and the mark bits of each of the objects it refers to are checked. If the mark bit is unset, the collector sets it and adds it to the work list. The black set in the abstraction consists of those objects that have been marked, and processed by the collector loop.

Once the mark phase is complete, the collector scans the heap sequentially, placing every object whose mark bit is zero (the final white set) on a list of free memory cells, and resetting the mark bit of every marked object to zero 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 and the work queue. Implementations vary

```
1  // Mark phase
2  for (object in root-set)
3    markBit(object).set()
4    work.add(object)
5
6  while (!work.isEmpty())
7    obj = work.remove()
8    for (child in referents(obj))
9      if (markBit(child).testAndSet())
10       work.add(child)
11
12 // Sweep phase
13 for (object in allObjects)
14   if (markBit(object).isSet())
15     markBit(object).clear()
16   else
17     free-list.add(object)
```

Figure 2.1: Mark-Sweep Garbage Collection

as to whether the mark bit is kept in the object header, or separately in a side array. Mark-sweep as described above is relatively inefficient, taking time proportional to the size of the whole heap regardless of the amount of live data. It also suffers from a large pause time, although there are concurrent mark sweep algorithms that reduce this. Naïve mark-sweep collection requires a free-list memory organisation, and therefore incurs a higher allocation cost and worse locality than garbage collection schemes that use monotonic allocation.

Hughes [1982] introduced the concept of *lazy sweeping*. Rather than sweep the entire heap during the garbage collection, the mutator sweeps the heap incrementally, generally a *block* at a time, where the definition of a block varies by implementation. While this does not decrease the total cost of sweeping, it does significantly reduce pause times. Boehm [2000] notes that lazy sweeping has a locality advantage, since the sweep operation loads memory into cache that the allocator is just about to use. When the cost of sweeping is delegated to the mutator, mark-sweep is the fastest full-heap garbage collection mechanism available. This advantage is offset by slow downs in the mutator. Since it requires a free-list allocator, mark-sweep suffers from poor allocation performance and poor mutator locality, and the mutator is further slowed down by the cost of lazy sweeping. We discuss performance aspects of lazy sweeping implementations in detail in Chapter 4.

2.1.3.2 Reference counting

Reference counting was the second garbage collection algorithm published, also in 1960 for the LISP system [Collins, 1960]. A reference counting collector extends each object header with a field that counts the number of pointers that refer to it. The mutator adjusts the reference count whenever a reference field is modified, incrementing the count of the new referent, and decrementing the count of the old. 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 transitively. 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 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. Lazy reference counting [Weizenbaum, 1963; Ritzau, 2003] eliminate these pauses by reclaiming freed data structures one object at a time in the mutator, but at considerable cost in space [Boehm, 2004].

One disadvantage of reference counting is that pointer mutations are frequent and the write barrier is expensive, especially in multi-threaded systems where synchronization on the reference count field is required. Thus straightforward reference counting implementations are generally among the slowest of memory management schemes. 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. Coalescing [Levanoni and Petrank, 2001] reduces the reference counting overhead of frequently mutated fields by recording only the initial and final state of pointer fields mutated between two garbage collection phases. Ulterior Reference Counting [Blackburn and McKinley, 2003] uses reference counting for the mature space of a generational collector, observing that pointer mutations of mature objects are much less frequent than younger objects, achieving a collector that combines the throughput of a generational collector with the low pause times of a reference counting collector.

The most significant disadvantage of reference counting is that it cannot in itself collect cyclic data structures, and alternative approaches such as *trial deletion* [Christopher, 1984; Martínez et al., 1990; Lins, 1992; Bacon and Rajan, 2001; Bacon et al., 2001] or mark-sweep collection (*backup tracing* [Deutsch and Bobrow, 1976]) 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 cause the object to be collected. The cost of trial deletion can have a significant effect on program run time [Quinane, 2003; Frampton, 2010], while using backup tracing to some extent obviates the need for reference counting at all. Frampton et al. [2008] and Frampton [2010] show how concurrent backup tracing can exploit the reference counting barriers to reduce overheads.

2.1.3.3 Evacuating

Evacuating collectors are a subclass of copying collectors, initially proposed by Hansen [1969] and Fenichel and Yochelson [1969], and first became practical with

Cheney's algorithm [Cheney, 1970]. Evacuating collection combines the advantage of low mutator overhead due to bump-pointer allocation 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 naïve mark-sweep collectors whose performance is proportional to the total size of the heap.

The original copying collector is known today as *semi-space*, 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 the live copy of the object is in to-space. When pointers are traced from live objects, either an object is still in from-space—in which case the collector copies it and updates the pointer to point to its new location, or it has already been copied—in which case the forwarding pointer is used to update the pointer being traced.

Cheney's algorithm uses the set of already-copied objects as a work queue to eliminate the need for an explicit work queue data structure, thus making copying a practical technique in LISP by bounding the space overhead required. In modern object-oriented and functional languages—where the average object size is significantly larger than a pointer—it is more common to use a separate data structure (e.g. a stack) for the work queue.

The attraction of copying collectors is that in typical programs few allocated objects survive to be collected, and hence the overhead of copying objects is small compared to the size of the heap. Copying collectors also permit the use of a bump-pointer allocation scheme which provides good allocation performance. The principal advantage of copying collectors, though, is that they compact the heap, improving locality for the mutator.

One disadvantage of pure copying collectors is that time is often wasted copying long-lived data structures from one semi-space to the other and back. A second disadvantage is that in order to allow for a worst-case collection where almost 100% of objects survive, it requires a 100% space overhead. Copying is generally used in modern generational collectors, where the ability to collect subsets of the heap independently allows copying to be targeted at object populations with low survival rates, where it performs best.

2.1.3.4 Compacting

Compacting is a subclass of copying collection that maintains the locality advantages of semi-space, but with almost no space overhead. The simplest form is *sliding* compaction, originally implemented in LISP-2 as a four phase algorithm [Styger, 1967]. The collector first performs a transitive closure, setting a mark bit for each object as in a mark-sweep collector. In the second pass, the collector calculates the future lo-

cation of each object, and writes a forwarding pointer to each object. In the third, or *forwarding* pass, the collector updates all pointers to reflect the addresses calculated in the second pass. In the fourth and final phase, the collector copies objects to their new locations. Copying is done in strict address order to ensure that no live data is overwritten.

While optimized versions of compacting collectors exist, they are still generally more expensive than the alternatives, and are rarely used as the primary collector in a high-performance system. More commonly, compacting is either used as a backup strategy for when space becomes tight [e.g. Sansom, 1991], or alongside a mark-sweep or mark-region collector to compact memory when fragmentation is detected. They do, however, have the advantage of excellent mutator locality because they preserve allocation order, and have very low space overheads.

2.1.3.5 Mark-region

The fifth canonical family of garbage collectors is known as *mark region*. These collectors allocate objects contiguously, and free it in small contiguous regions when all objects in the region are unreachable. The most significant examples on this class of collector are Immix [Blackburn and McKinley, 2008] and the original JRockit collector[1]. Immix allocates 32KB *blocks*, and allocates contiguously within these blocks. During collection, each live object is marked with a bit, as with mark-sweep collectors. In addition, a second bit is set for each 128-byte *line* in which the object resides. After collection, all blocks are placed in the queue for allocation. The allocator uses the per-line mark bits to identify free space within the block, and allocates contiguously into these regions.

Mark Region collectors combine excellent allocation performance and good collector performance, with very low space overheads and good mutator locality.

2.1.3.6 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. Observation shows that the weak generational hypothesis holds for many workloads, whereas in general the strong generational hypothesis does not [Clinger and Hansen, 1997].

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

[1] The JRockit garbage collector was documented in a web page that is no longer available. Secondary citations for it can be found in several papers, including [Hallberg, 2003; Blackburn and McKinley, 2008].

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 a *remembered set* for use during minor collections. The space containing the oldest objects in a generational collector is known as the *mature space*.

Different collection policies can be applied to each generation in a generational collector. For example, Jikes RVM's production configuration uses a generational collector with an evacuating nursery and an Immix mature space. The default collector in the Sun (Oracle) hotspot virtual machine uses an evacuating nursery, a pair of semi-spaces as the second generation, and a mark-sweep mature space. One significant style of generational collector is the Appel-style collector [Appel, 1989]. This is a two-generation collector, where the evacuating nursery is allowed to use 50% of the available free space, shrinking as the mature space grows and then growing again after a full-heap collection. By making maximum use of the available space for nursery collections, the throughput of the collector is maximised.

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

2.1.4 Object Scanning

The transitive closure operation of a garbage collector requires locating the references from the stacks and other structures outside the heap (the roots), and then finding the referents of a heap object, an operation we refer to as *scanning* the object. Broadly speaking there are two ways to locate references in each of these domains, as well as a hybrid approach.

Conservative collectors such as the Boehm-Demers-Weiser (BDW) collector [Boehm and Weiser, 1988] use heuristics to identify which words in the stack and other non-heap areas could possibly be heap pointers. Using these values as roots, a conservative collector traces the heap, regarding all words as potential pointers. The BDW collector uses a segregated free-list with a large number of closely packed size classes, and can therefore identify the boundaries of objects given a pointer into the heap. Because a conservative collector is never sure which values are actually pointers rather than (for example) character sequences or integers, the collector must never move objects, and the BDW collector uses a mark-sweep algorithm. The principal concern of the object scanning implementation in a conservative collector is in designing heuristics that distinguish references from non-references.

The other end of the design spectrum is a *precise* collector, which uses information generated by the compiler and runtime environment to identify reference fields precisely. Garbage collection is only permitted to occur at specific *safe points*, and at each such point the compiler generates a *gc map* of the stack frame to identify the live heap references. The run-time system maintains a map of which global (static) fields are heap references, and also metadata that identifies which fields within heap objects contain references—in a Java virtual machine this metadata is generated by the class loader. The principal concern of the object scanning implementation in a precise col-

lector is in efficiently accessing and using the metadata that describes the layout of each object.

In the design space between precise and conservative collectors are *ambiguous roots* 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. Bartlett's mostly copying collector takes advantage of this by using copying techniques for objects in the heap that are not pointed to by the (ambiguous) root set.

The other hybrid approach—precise roots and conservative object scanning—is possible but uninteresting in real-world settings. The implementation effort to implement precise stack scanning is significantly greater than that of precise heap scanning, so in systems where the former is in place it makes little sense to omit the latter.

2.2 Infrastructure

This section describes the infrastructure used to implement the techniques developed in the later chapters of this thesis. This infrastructure is integral to this thesis, and throughout the course of my PhD I made substantial contributions to the design, maintenance and improvement of Jikes RVM and MMTk, the systems I now describe.

2.2.1 Jikes RVM

Jikes RVM [Alpern et al., 2000] is an open source high performance Java virtual machine (VM) written almost entirely in a slightly extended Java [Alpern et al., 1999], and is the basis for all concrete implementations presented in this thesis.

Besides being written in Java, Jikes RVM is distinctive in that it uses a compile-only strategy, and as such does not have a bytecode interpreter. Instead, a fast template-driven baseline compiler produces machine code when the VM first encounters each Java method. The adaptive compilation system then judiciously optimizes the most frequently executed methods [Arnold et al., 2000]. Using a timer-based approach, it schedules periodic interrupts. At each interrupt, the adaptive system records the currently executing method. Using a threshold, it then selects frequently executing methods to optimize. Finally, the optimizing compiler thread re-compiles these methods at increasing levels of optimization.

In order to 'tie the knot' of meta-circularity, the virtual machine portion of Jikes RVM is compiled ahead of time using its own compiler, saving the resulting machine code in a *boot image*. The memory manager is part of this boot image, and therefore enjoys the benefits of being compiled using the Jikes RVM optimizing compiler while not incurring any compilation cost at runtime. The optimizing compiler options chosen when building the Jikes RVM boot image have a significant effect on the performance of the resulting virtual machine. We use a 'FastAdaptive' build, which compiles the boot image with maximum optimization and disables assertions. In recent versions

of Jikes RVM, the boot image can be compiled using a *profiled build*, which runs the unoptimized virtual machine briefly to generate edge counter information which is then used by the compiler to build the final optimized build.

2.2.1.1 Low-level Programming

Frampton, Blackburn, Cheng, Garner, Grove, Moss, and Salishev [2009] describe in detail the approach (known as *vmmagic*) used by Jikes RVM to break the Java language's abstractions and deal directly with the underlying hardware. This provides several features essential to the implementation of a virtual machine, including *unboxed types* on which much of the work described in later chapters relies.

Unboxed types as implemented by vmmagic allow us to declare local variables and object fields which are stored directly (like an `int` or `long`, and unlike an `Object`), but which have methods. One such unboxed type is *Address*, which represents an address in memory, and which has the size of the underlying hardware address. Methods of an *Address* include `loadInt()`, which loads a 32-bit integer at the represented address. The compiler compiles this as an *intrinsic*, substituting an IR sequence that (if not optimized away) eventually translates to a single instruction rather than a full method call.

Using vmmagic gives us 'abstraction without guilt'—the ability to write fast code without sacrificing too much of the expressive power of the high-level language we develop in.

2.2.1.2 Replay Compilation

One feature of Jikes RVM that we use extensively in this thesis is *replay compilation* [Huang et al., 2004]. This allows us to use the benefits of adaptive optimization but without the experimental 'noise' that is inherent in an adaptive timer-based optimization process.

Replay compilation requires running an experiment in two different modes. During the first execution, each benchmark is run for a number of training iterations until it has reached a stable state, and the adaptive recompilation system has compiled all the 'hot' methods. On exit, the replay compilation system saves the list of optimized methods, their levels of optimization and the profile information used by the optimizing compiler to compile the methods.

The second and subsequent executions are used to measure performance. During these executions, the benchmark runs for two iterations. During the first iteration, the compilation data recorded during the first execution is used to compile methods to their final optimization level the first time they are invoked. By the end of the first iteration of the benchmark, all methods have been optimized to the level they were at the end of the final iteration of the first execution, and the replay system disables the compiler and adaptive optimization systems. We time the second iteration of the benchmark, which runs with the benefit of fully optimized code but without interruption by the compiler.

When running several invocations of a benchmark to measure experimental error, we use the same replay data. Ensuring that a benchmark is compiled the same way for each invocation significantly reduces experimental error introduced by the adaptive compilation system.

When comparing several different runtime system optimizations, we can also use the same replay compilation data, ensuring that the mutator's behaviour is common across the optimizations being compared.

2.2.2 MMTk

MMTk is Jikes RVM's memory management sub-system. It is a composable memory management toolkit that implements a wide variety of collectors that reuse shared components [Blackburn et al., 2004a].

MMTk collects together coherent sets of components into memory management *plans*, and when a virtual machine invokes MMTk, it selects one of these plans to manage its heap.

A plan in MMTk divides up virtual memory by *policy*, allowing different areas of the heap to be managed in different ways. A typical plan may provide a *large object space* managed using Baker's Treadmill [Baker, 1992], an Appel-style copying nursery [Appel, 1989] and a mark-sweep mature space.

The experimental work presented in this thesis uses MMTk's MarkSweep plan, a full-heap collector in which objects smaller than 8KB are managed by a mark-sweep policy[2].We do this for a number of reasons:

1. Some of the optimizations we implement only apply to non-copying collection.
2. MarkSweep has the fastest tracing rate of any of MMTk's non-copying collectors. This tends to display more prominently the results of any optimizations.
3. Mark-sweep is a well known algorithm with a long history. It is representative of a much larger class of collectors.

Many of the contributions presented in this thesis have been integrated into MMTk.

2.2.2.1 The MMTk Free-list Allocator

The segregated free list allocator used by MMTk's MarkSweep space maintains separate free lists for 40 different size classes. When a free list for a given size class is empty, one to three 4KB-blocks of memory is allocated and split into cells of the free list size. Block sizes are tuned to ensure that a maximum of 14% internal fragmentation occurs in any given size class, and total fragmentation is typically much less than this.

At the block level, each active thread maintains one private block for each size-class it has allocated since the previous garbage collection. At the start of each GC, all

[2]Objects larger than 8KB are managed by a specialized Large Object Space using Baker's treadmill [Baker, 1992], allocating space using 4KB pages.

Platform	Jikes RVM	GCJ 4.0
AMD Athlon64 3500+	266	194
Pentium-4 *Prescott*	265	214
Pentium-M *Dothan*	210	189
PowerPC 970 G5	218	291

Table 2.1: Comparative GC performance, MMTk (Jikes RVM) vs. Boehm (GCJ). GC Throughput in MB/s

blocks are returned to the global pool and at the start of the next mutator cycle they must acquire new blocks from the global pool. This helps to minimise the amount of space tied up in per-thread blocks.

2.2.2.2 Credibility of MMTk As An Experimental Platform

In our prefetch paper [Garner et al., 2007] we compare the performance of the tuned version of MMTk used in this paper with gcj using the standard Jikes RVM GC performance benchmark, FixedLive. Gcj [GCJ] is an ahead-of-time Java compiler which uses the Gnu Compiler Collection back-end and the Boehm conservative garbage collector [Boehm and Weiser, 1988; Boehm, 2012]. The Boehm collector is highly tuned for this environment, with a great deal of tuning in gcj aimed at reducing the root set to the smallest memory regions containing pointers into the heap. The FixedLive benchmark starts by allocating a large binary tree which lives for the duration of the benchmark, and then allocates a large number of very short-lived objects, using heuristics to identify and time five garbage collections. The results of this comparison are given in Table 2.1, with the numbers representing a tracing rate in MB/s. This shows that MMTk outperforms gcj by between 11% and 37% on the x86 platforms, but lags gcj on the PPC by 33%. The tracing performance of gcj on the PPC is considerably faster than on the x86 architectures, despite the fact that on most benchmarks the PPC 970 is slower than the other machines we used. We believe that the prefetching in the BDW collector is much more effective on the PPC, and that this allows it to outperform MMTk with no prefetching.

This comparison uses the same baseline MMTk MarkSweep configuration that is the basis of Chapter 4. We use gcj version 4.0.2, and compile using the '-O2' flag. The BDW collector is a conservative (ambiguous roots) collector, thus some of the performance difference may be due to it being unable to take advantage of the fast object scanning techniques used by MMTk. Our goal here, however, is simply to assert that MMTk is a well tuned platform and a credible basis for experimentation, relative to prior work.

2.3 Evaluation Methodology

When evaluating the performance of computer systems, researchers must perform this evaluation in the context of a defined workload or *benchmark*. This section looks

at the issues of benchmarking and evaluation of results as they pertain to the work presented in the body of this thesis. Like the infrastructure I use, the benchmarks are fundamental to this thesis. I was deeply involved in the DaCapo benchmark project, as a lead author, and maintainer and developer of its evaluation framework.

2.3.1 Benchmarks

> "The best choice of benchmarks to measure performance are real applications, such as a compiler. Attempts at running programs that are much simpler than a real application have led to performance pitfalls. Examples include
>
> - kernels, which are small, key pieces of real applications;
> - toy programs, which are 100-line programs from beginning programming assignments, such as quicksort; and
> - synthetic benchmarks, which are fake programs invented to try to match the profile and behavior of real applications, such as Dhrystone.
>
> All three are discredited today, usually because the compiler writer and architect can conspire to make the computer appear faster on these stand-in programs than on real applications."
>
> Hennessy and Patterson [2006, p. 29]

Performance-oriented work on computer systems aims to improve the performance of real-world applications. While some work is targeted at a specific application or class of applications, other work targets a much wider domain. The difficulty when evaluating research is that many applications are hard to run repeatedly in controlled conditions. Many have long run-times, or require a complex hardware environment, or are licensed proprietary code. The accepted practice is to use *benchmarks*, a carefully constructed or selected set of applications with defined workloads. Performance results against a well designed suite of benchmarks will be indicative of performance on the real-world programs that are the ultimate target of the research. Conversely, a bad choice of benchmarks may lead to misleading or incorrect results.

The performance analysis in this work is performed using the SPECjvm98, SPECjbb2000 and DaCapo benchmarks. This section describes these benchmarks in more detail.

2.3.1.1 The SPEC Benchmarks

When evaluating Java virtual machines, industry and academia typically use the SPEC Java benchmarks SPECjvm98 and SPECjbb2000 [SPEC, 1999, 2001]). The SPECjvm98 suite contains benchmarks derived from real programs, although by 2006 the benchmark suite was distinctly outdated. These have now been superseded by SPECjvm2008 and SPECjbb2005 [SPEC, 2008, 2006], but for researchers these suites still have

some significant drawbacks. The SPEC benchmarks are primarily targeted at producing a single number that can be used for comparisons between computer systems. With the SPECjvm suites, the SPEC benchmark harness makes it possible to run and measure individual benchmarks, however some modification to the suite is required to (for example) time the *n*th iteration, or start and stop performance counters.

SPECjbb has a more fundamental problem for memory management research. In both the 2000 and 2005 versions, the benchmark runs for a fixed amount of time and reports the number of transactions performed per unit of time. While a performance improvement can be measured as it contributes to the overall throughput, it becomes impossible to directly measure improvements in garbage collector time—the better the garbage collector performs, the more work the benchmark performs, and hence the more garbage collection it performs. For these reasons, when using SPECjbb we use pseudojbb, a fixed-workload version.

In this thesis I use SPECjvm98 and SPECjbb2000, because at the time of performing the experiments, Jikes RVM was unable to run SPECjvm2008 and SPECjbb2005 due to limitations in its class libraries.

2.3.1.2 The DaCapo Benchmarks

The DaCapo benchmarks were designed and built by a large group of researchers, in order to address some of the shortcomings in the available Java benchmark suites. The first official release of the DaCapo benchmarks was made in October 2006, to coincide with the publication of our OOPSLA paper [Blackburn et al., 2006]. The DaCapo benchmarks are substantially more complex and varied than the SPEC benchmarks. The benchmarks were chosen with several goals in mind: relevance—all of the benchmarks should be nontrivial, actively maintained, and used in real-world settings; diversity—the benchmarks should cover a range of problem domains and coding styles; and suitability for research—a controlled, tractable workload amenable to analysis and experiments [Blackburn et al., 2008].

2.3.2 Evaluation Methodology

Evaluation methodology for memory management research in a virtual machine environment is a current topic of debate in the programming language research community. The evaluation methodology used in the subsequent chapters of this thesis builds on the work in Blackburn et al. [2006, 2008].

The key features of our evaluation methodology are:

1. **Wide selection of quality benchmarks.** We evaluate on at least 17 benchmarks from the SPECjvm98, DaCapo and SPECjbb2000 suites. Only where unavoidable, we subset the benchmarks, and clearly identify where and why.

2. **Control for non-determinism.** We use replay compilation (Section 2.2.1.2) to provide a deterministic compilation plan. We also run experiments on a dedicated machine without any non-essential background tasks.

3. **Multiple hardware platforms.** All of our results are produced on 4 or more hardware platforms. This produces some results that are of interest because they are hardware-specific, while protecting against the pitfall of generalising from a single quirky platform.

   Different portions of the thesis are evaluated on different sets of platforms. This is an unavoidable consequence of having performed the experimental work for the thesis over a period of six years. All the hardware platforms used were operated in 32-bit mode, since Jikes RVM did not have a mature 64-bit compiler implementation for the Intel architecture at the time.

4. **Multiple iterations.** Chapter 4 runs experiments with 5 iterations of each benchmark and takes the fastest of the 5 runs. Chapter 5 uses 6 iterations, takes the mean of all iterations and shows a 90% confidence interval calculated using Student's t-distribution. Chapter 3 uses a mix of these two methods. Since the results in Section 3.3.1 and Section 3.3.2 are used to support the contributions in Chapter 4 they use the methodology of that chapter, while Section 3.3.3 was performed later and uses the more recent methodology of Chapter 5.

Our own advice in Blackburn et al. [2006] is to use multiple virtual machines. It is the nature of experimental software implementation to be extremely time consuming, and implementing our experiments on a virtual machine other than Jikes RVM would be prohibitive given the available time, so where applicable we explicitly identify aspects that are specific to the virtual machine.

Garbage collector performance is in general a time/space trade-off, and it is usual to present results using a range of heap sizes. The results presented here affect the performance of basic garbage collection mechanisms, and are essentially independent of heap size. For this reason most of our results are presented for a single heap size, although we take care to verify this assumption for each optimization.

Georges et al. [2007] demonstrate the potential importance of reporting the mean with a calculated confidence interval when evaluating Java performance on modern virtual machines. As mentioned above, the work in Chapter 4 uses the minimum of a set of results and calculates no confidence interval. While it would be straightforward (if time consuming) to recalculate the results, the experiments in Chapter 5 do use these calculations and find the experimental error to be extremely low. For this reason we believe that the results in Chapter 4 would hold if we were to recalculate them.

2.4 Summary

This chapter gave an overview of garbage collection, and the benchmarks and infrastructure used in the experiments that follow. Subsequent chapters will introduce novel evaluation and implementation techniques that allow us to understand the behaviour of benchmarks, and improve the performance of garbage collectors.

Chapter 3

Garbage Collector Performance

The previous chapter gave background information about garbage collection, the benchmarks used to evaluate performance, and the infrastructure we use for our experiments. This chapter presents a new technique—which we call *replay tracing*—for analysing the performance of a garbage collector. We build on the performance results and insight obtained using replay tracing in later chapters.

This chapter is based in part on work published in the paper "Effective Prefetch for Mark-sweep Garbage Collection" [Garner, Blackburn, and Frampton, 2007]. The key contribution of this work is the technique of replay tracing, and performance analyses we present.

The chapter is structured as follows. Section 3.1 provides a brief introduction to the problem of performance evaluation of the garbage collector tracing loop and describes the prior work in this area. Section 3.2 describes the replay tracing framework in detail. Section 3.3.1 analyses the tracing loop using the replay tracing framework, describing the experiments we use to determine the costs of the various parts of the loop. Section 3.3.2 uses replay tracing to evaluate two possible implementations of mark state, a side bitmap and header mark state. Section 3.3.3 looks at the performance impact of heap traversal order, using replay tracing to ensure that implementation details of the queue structure are removed from the equation. Finally, Section 3.4 summarises the results and shows how they inform later chapters of the thesis.

3.1 Introduction

The tracing loop is the most performance-critical element of any garbage collector, and in particular it is the portion that scales with the volume of live memory in the heap. Table 3.1 shows the percentage of GC time spent tracing the heap on the machines used in Section 3.3.3. We can see from this table that the tracing loop accounts for between 49% (compress on the AMD Phenom) and 96% (hsqldb on the Atom D510) of the GC time. Previous work has used sample-based profiling [Boehm, 2000] and simulation [Cher et al., 2004] to analyze the mechanism, each of which have shortcomings. Simulation has the disadvantage of long running times, making it difficult to use large, realistic benchmarks. Simulation also limits the available target architectures to those supported by the simulation packages available, and is entirely dependent on the fi-

	atom	core2	corei5	phenom
compress	59%	50%	56%	49%
jess	64%	57%	63%	57%
raytrace	68%	59%	67%	61%
db	68%	61%	65%	60%
javac	65%	59%	64%	58%
mpegaudio	62%	53%	59%	52%
mtrt	70%	63%	71%	65%
jack	61%	54%	60%	53%
pseudojbb	90%	90%	91%	89%
antlr	75%	70%	74%	69%
bloat	78%	75%	78%	75%
fop	82%	79%	83%	80%
hsqldb	96%	94%	95%	95%
jython	85%	82%	85%	82%
luindex	71%	67%	71%	67%
lusearch	71%	67%	71%	66%
pmd	85%	83%	86%	83%
xalan	81%	78%	82%	79%
Min	59%	50%	56%	49%
Max	96%	94%	95%	95%

Table 3.1: Tracing loop time as a percentage of total GC time.

delity of the simulation infrastructure with respect to real hardware. Sample-based profiling limits analytical flexibility: in order to sample a collector, it must be a working real-world collector; this makes it time consuming to experiment with algorithmic and implementation variations, and very hard to tease apart the contributions of various details of the implementation. Furthermore, sampling is inherently probabilistic rather than exact.

Hicks et al. [1997] present what is probably the work closest to our approach. Their system instruments a language runtime in order to take a snapshot of the heap, and use a special garbage collector evaluation tool to replay a traversal of the heap. This has the advantage of requiring even less of a runtime than our system. The disadvantage of their system is that examining GC behaviour over the entire run of a benchmark requires taking a snapshot at each GC, producing extremely large trace files in some cases. Our approach overcomes these limitations and is more light-weight, with the basic replay tracing infrastructure requiring only a few hundred lines of code.

3.2 The Replay Tracing Framework

The solution presented here is called *replay tracing*. Our initial implementation uses a modified mark-sweep garbage collector. The system works by modifying the garbage collector so that at *every* collection, in addition to performing collection work, the collector gathers a trace of visited objects and then *replays* and measures that trace multiple times for analytical purposes. This approach allows experimentation with a great many variations on the tracing loop, and by using timers and hardware performance counters we can analyze the various costs in detail.

At each collection, we first trace the live objects in the heap, exactly as the unmodified mark-sweep collector would, except that whenever an object is processed

(popped from the mark stack), we record a pointer to the object in a *replay buffer*. The replay buffer gives us a record of the objects accessed during the trace, in exactly the order in which they are accessed. We then use the replay buffer to execute multiple replay *scenarios*. Each scenario performs different operations on every object in the buffer. The objects are processed in exactly the same order for each scenario.

For example, a scenario which just performs a mark operation on each object allows us to isolate the cost of marking and thus evaluate different marking strategies. Likewise, a scenario could just touch each object, scan each object, or perform a complete mark, scan and trace of each object. By carefully constructing scenarios and measuring their costs, we can break down the contributions of the various elements of the tracing loop and systematically explore alternatives.

In order to minimize distortion of results due to cache pollution, we flush the cache between each use of the replay buffer by reading a large table sequentially. We use a table at least 4 times the size of the last-level cache. We also need to take care when setting mark bits—their state must be flipped after each phase in which they are changed. We repeat all of this—creating the replay buffer, replaying scenarios, and flushing the cache—each time a collection is triggered. We aggregate results across collections so that at the end of the program we have measurements for each scenario with respect to the entire GC workload of the program.

3.3 Results

We now use the framework to conduct three studies. In Section 3.3.1 we study the costs of the tracing loop. In Section 3.3.2 we evaluate the performance of side versus header metadata. In Section 3.3.3 we evaluate the impact of traversal order on the performance of garbage collection.

3.3.1 The Composition of the Tracing Loop

Our first objective of replay tracing is to break down the cost of the transitive closure operation, the *tracing loop*, into its component parts. Once we know which parts of the loop are most expensive, we can target them for optimization.

3.3.1.1 Replay Scenarios

We now describe a number of example replay scenarios, including those we use in our subsequent analysis.

Figure 3.1 shows pseudo-code for the standard mark-sweep tracing loop, from which most of the scenarios in Figure 3.2 are derived. We have described the queue operations that maintain the *work list* abstractly as *add* and *remove*. If a collector implements these operations as push and pop (LIFO–a stack), it will perform a depth-first traversal of the heap graph, while tail-insert and pop (FIFO–a queue) will produce a breadth-first traversal. The major components of the basic tracing loop in Figure 3.1 are: i) queuing costs (lines 4, 6, 7 & 13), ii) accessing the reference map for each object

```
1  for p in root-set
2    obj = p.load()
3    mark(obj)
4    queue.add(obj)
5
6  while !queue.isEmpty()
7    obj = queue.remove()
8    gcmap = obj.getGcMap()
9    for p in gcmap.pointers()
10     child = p.load()
11     if child != null
12       if child.testAndMark()
13         queue.add(child)
```

Figure 3.1: The Standard Tracing Loop

(line 8), iii) enumerating reference fields (line 9), and iv) the test and mark of referenced objects (line 12).

In order to evaluate the relative costs of these operations, we use the scenarios shown in Figure 3.2. The *harness cost* scenario in Figure 3.2(a) measures the cost of the replay buffer and thus the overhead of our framework. The *queue cost* scenario in Figure 3.2(b) measures the approximate queue management cost of the standard tracing loop, alternately inserting N items onto the queue and popping $N-1$ items (we use $N = 10$) so that the effects of growing and shrinking the queue across block boundaries are measured. Once all items are inserted, the remaining items are popped from the queue. For this scenario we use MMTk's standard `Deque` data structure as a stack, i.e. the same way MMTk uses it in Jikes RVM's production collectors.

The *object touch* scenario (Figure 3.2(c)) measures the cost of accessing the first word of each reachable object in the heap.[1] The GC map, describing the location of any reference fields within the object, is typically only found via touching (and possibly dereferencing) the header of the object to be scanned. The *scan* scenario in Figure 3.2(d) measures the cost of visiting each object in the heap, iterating its GC map, and loading each reference field. Comparing the cost of this scenario with a scenario that simply visits each heap object can tell us about the incremental cost of the scan operation. The *trace* scenario in Figure 3.2(e) adds to the *scan* scenario a dereference of every non-null child of the scanned object. Finally, the *mark* scenario Figure 3.2(f) performs a mark on each non-null child of every object in the replay buffer. The *mark* scenario thus performs all of the work of the standard tracing loop except the final enqueuing operation; compare lines 9 to 13 in Figure 3.1 with lines 4 to 7 in Figure 3.2(f).

One might be tempted to assume that the change in workload of two scenarios where the second scenario strictly adds work to the first one can be measured by simple subtraction. This is true for wall clock time (at least in our observations), but

[1] In Jikes RVM, the object pointer actually points 4 bytes past the first field of a scalar object, and at the first element of an array. It is this word that we access.

```
for item in buffer
  item.load()
```

(a) Harness Cost Scenario

```
i=0
for item in buffer
  queue.add(item.load())
  if ++i == N
    while --i > 0
      queue.remove()
    i = 0
while !queue.isEmpty()
  queue.remove()
```

(b) Queue Cost Scenario

```
for item in buffer
  item.load().load()
```

(c) Object Touch Scenario

```
for item in buffer
  obj = item.load()
  gcmap = obj.getGCMap()
  for p in gcmap.pointers()
    child = p.load()
```

(d) Scan Scenario

```
for item in buffer
  obj = item.load()
  gcmap = obj.getGCMap()
  for p in gcmap.pointers()
    child = p.load()
    if (child != null)
      child.load()
```

(e) Trace Scenario

```
for item in buffer
  obj = item.load()
  gcmap = obj.getGCMap()
  for p in gcmap.pointers()
    child = p.load()
    if (child != null)
      child.testAndMark()
```

(f) Mark Scenario

Figure 3.2: Replay Scenarios

not necessarily true for other measures such as cache misses. In fact, the *object touch* scenario (Figure 3.2(c)) that visits the first word of each object in the heap actually has a higher L2 miss rate than the *scan* scenario (Figure 3.2(d)), on some architectures, even though the scan scenario does strictly more work (`obj.getGCMap()` involves at least one `load()`). We suspect that the additional work in the scan scenario is allowing a hardware prefetch mechanism time to take effect, eliminating some of the misses seen by the first scenario.

This non-additivity illustrates one of the pitfalls of using microbenchmarks for performance evaluation (since the replay scenarios can be seen as microbenchmarks), and of relying on secondary measures of performance. These results are interesting because of the directions they suggest for performance improvement, rather than the absolute performance figures obtained. Any improvement we make based on these figures is again tested in the context of the full loop, using wall-clock time as the performance measure.

Platform	Clock	DRAM	L1 cache	L2 cache
AMD Athlon 64 3500+	2.2GHz	400MHz DDR2	64KB 64B 2-way	512KB 64B 16-way
Pentium-M *Dothan*	2.0GHz	533MHz DDR2	32KB 64B 4-way	2MB 128B 8-way

Table 3.2: Hardware Platforms used for the Replay Tracing Loop Costs Experiments

3.3.1.2 Tracing Costs

We now use the replay tracing framework described in Section 3.2 to evaluate the dominant costs of the mark-sweep tracing loop shown in Figure 3.1. We perform our measurements using Jikes RVM and MMTk with DaCapo and SPEC benchmarks, as described in Chapter 2, and the AMD and Pentium-M architectures shown in Table 3.2.

Framework Overhead We begin by measuring the overhead of the replay tracing harness. We compare the cost of the simple *harness cost* scenario shown in Figure 3.2a against the cost of a full collection of the heap. We found that in terms of wall clock time, the cost of the harness was less than 2% that of a full heap collection. Hardware performance counters revealed negligible L1, L2 and DTLB misses, while the harness accounted for 6% of the instructions executed.

Experiments To analyze the cost of the mark-sweep tracing loop, we use the methodology described in at the beginning of the chapter, evaluating a series of incrementally more complex replay scenarios to build up a picture of total costs. The replay scenarios we use are as follows:

enq-deq Enqueue/dequeue operations in the ratio 10:9, as reflected by the *queue cost* scenario in Figure 3.2b. This identifies the cost of the queuing mechanism on which the standard tracing loop is based. All of the subsequent scenarios do not use a queue, because they are driven by the replay buffer, which captures and replays an exact object visit order (see Section 3.2).

touch Load the first word of every object pointed to by the trace buffer, as reflected by the *touch* scenario in Figure 3.2c. This identifies the cost of touching every live object in the heap. In the standard tracing loop, the first step after obtaining a new reference to work on is to fetch the GCmap for the object (line 7 of Figure 3.1). In Jikes RVM (and many other Java virtual machines) this involves fetching a per-class data structure pointed to by a word in the object's header. By loading the first word of each object, we access memory in a pattern very similar to that of the first action in fetching a GC map. Note that this scenario does not place any dependencies on the load, so the costs of the loads may be understated by wall clock time. However, this scenario will help understand the locality properties of a full heap trace.

	Sync	Unsync
Traverse	0.02	0.03
Enq-deq	0.10	0.10
Touch	0.15	0.15
Scan	0.40	0.40
Trace	0.59	0.59
Mark	**1.00**	0.84

(a) Pentium-M

	Sync	Unsync
Traverse	0.02	0.02
Enq-deq	0.11	0.11
Touch	0.14	0.14
Scan	0.46	0.46
Trace	0.63	0.63
Mark	**1.00**	0.89

(b) AMD Athlon64

Table 3.3: Elapsed Time for Various Scenarios for Two Design Points, Normalized to the Synchronized Mark Scenario.

scan Scan every object in the trace buffer, loading the value of each of its pointer fields, as shown in Figure 3.2d. This builds on the *touch* operation by using the GC map to enumerate the pointer fields in the object and load the value of each pointer.

trace Trace 'through' every object in the trace buffer, *dereferencing* each of its non-null pointer fields, as shown in Figure 3.2e. This builds on the *scan* operation by accessing the word pointed to by each pointer field in the object, i.e. touching each object referenced by the current object. This will perform one additional memory access per non-null pointer than the *scan* scenario. In an implementation where mark bits are kept in the header of an object, this operation will access memory in an analogous pattern to *mark*, but without writing the updated mark state.

mark Perform `testAndMark()` on every non-null child of every object in the replay buffer. If mark state is implemented in the header (see Section 4.3), this scenario only differs from the *trace* scenario by using a `testAndMark()` on each object's header, rather than a `load()`. If mark state is implemented in a bitmap on the side, this scenario will not touch the child object, and therefore is similar to the *scan* scenario, differing only in that it touches the side bitmap associated with the child. In either case, this scenario differs only from the full tracing loop in that unmarked pointers are not enqueued for later tracing.

3.3.1.3 Results

We now present a detailed analysis of the tracing costs for two variations of the standard marking mechanism, where mark state is set with a simple store (*unsync*) or with architecture-specific atomic update instructions (*sync*). Mark state for all these experiments is in the header of the object. Table 3.3 shows elapsed times for each of the scenarios for each variant, on the Pentium-M and AMD platforms from Table 3.2, with all data normalized to the *sync* time on the respective platform. Recall that the *mark* scenario only differs from the full tracing loop in that it does not enqueue the marked

	Sync				Unsync			
	Time	RI	L1	L2	Time	RI	L1	L2
Traverse	0.02	0.06	0.00	0.00	0.03	0.06	0.00	0.00
Enq-deq	0.11	0.30	0.06	0.01	0.13	0.31	0.06	0.01
Touch	0.15	0.10	0.45	0.59	0.18	0.10	0.45	0.58
Scan	0.39	0.54	0.53	0.43	0.47	0.55	0.53	0.43
Trace	0.59	0.63	0.98	1.04	0.71	0.65	0.97	1.02
Mark	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**

(a) Pentium-M

	Sync				Unsync			
	Time	RI	L1	L2	Time	RI	L1	L2
Traverse	0.02	0.06	0.00	0.00	0.02	0.06	0.00	0.00
Enq-deq	0.11	0.30	0.06	0.01	0.12	0.31	0.06	0.01
Touch	0.14	0.10	0.51	0.54	0.15	0.10	0.51	0.54
Scan	0.46	0.54	0.58	0.56	0.51	0.55	0.58	0.56
Trace	0.63	0.63	0.99	1.01	0.71	0.65	0.99	1.01
Mark	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**

(b) AMD Athlon64

Table 3.4: Costs for Two Designs, Showing Time, Retired Instructions, L1 & L2 Misses, Normalized to Each Mark Scenario.

child (compare Figure 3.1 lines 7 to 11 and Figure 3.2(f) lines 3 to 7). The *mark* scenario thus provides a reasonable baseline. Table 3.4 includes and expands upon this data by providing retired instructions (RI), L1 cache misses, and L2 cache misses. In Table 3.4 the data is normalized against the *mark* scenario for each respective implementation variant (i.e. each column is normalized to the bottom row).

A number of observations can be drawn from Tables 3.3 and 3.4:

1. Synchronization accounts for approximately 17% of the cost of tracing the heap on a Pentium-M, and 11% on the AMD Athlon. This is evident when comparing normalized *mark* times for *sync* and *unsync* columns in Table 3.3. Unsurprisingly, this shows that designing data structures to avoid synchronization is a worthwhile goal, given current hardware implementations.

2. The overhead of queue management (enq/deq) is at most 13% of the running time of the tracing loop (Table 3.4).

3. *Touch* data in Table 3.4 shows that the initial visit of an object header accounts for 14–18% of time, but 45–60% of cache misses. This is perhaps accounted for by a lack of any dependency on the load, and combination of compiler instruction scheduling and out-of-order execution overlapping the initial fetch of an object with the mark operation on the previous object. This result helps to illuminate the limited success that other approaches have had with prefetch strategies that target this aspect of the tracing loop—even halving these misses would not yield a significant performance gain.

4. Comparing *scan* and *touch* L2 numbers in Table 3.4 shows that cache miss rates are non-monotonic; scenarios that perform more work sometimes have lower cache miss rates than simpler scenarios. Note that time and RI numbers remain monotonic. We suspect that this is accounted for by hardware prefetching—in particular the *scan* scenario performs a great many register/ALU operations in between memory fetches, allowing time for prefetched cache lines to arrive.

3.3.2 Mark State Implementations

The collector's mark phase depends on being able to store mark state for each object. Mark state is typically stored either as a field within the *header* of each object, or in a dense *bitmap* on the side. We refer to these two approaches as *header* and *side bitmap* respectively. The side bitmap has a potential locality advantage because it is far more tightly packed than mark bits embedded within headers of objects which are dispersed throughout the heap. While we refer to the mark bitmap as though it were a single contiguous data structure, in practice it is broken into chunks and distributed through memory. MMTk keeps metadata for each 4MB region of memory at the start of the region.

In a parallel collector, it is possible to lose updates in a race to set a bit unless the bit can be set atomically. An alternative to side bitmaps is a *bytemap*, a byte-grained data structure on the side. Bytemaps trade an $8\times$ space overhead to avoid synchronizing on each set operation, if we assume support for atomic byte-grained stores. An unsynchronized test-and-mark operation on a bytemap or a header field risks marking and scanning an object twice. In a non-moving collector there is no correctness issue—the worst possible consequence is that a small amount of work is duplicated between collector threads. Assuming an 8-byte minimum object size, an address-mapped bytemap would impose a 12.5% space overhead.

We include both a side bitmap and the header approach for comparison in this section. We don't evaluate a bytemap, since it has similar locality characteristics to the bitmap, and we expect it to perform very similarly to the unsynchronized bitmap.

To evaluate this design choice, we repeat the experiments in Section 3.3.1.2, adding a side bitmap as an additional dimension. Table 3.5 expands the results in Table 3.3 to include this data.

One of the reasons why mark state is often stored in side bitmaps rather than in object headers is the locality advantage of greatly increased density of the mark state (Section 4.3 and [Boehm, 2000]). It seems intuitively likely that a denser data structure will lead to lower cache misses than a scheme where mark bits are held in object headers and distributed over the whole heap. However, comparing the numbers in Table 3.5 shows that there is no significant performance difference. The overhead of synchronization dominates, and when synchronization is not used, the difference is only 1%, with one result in either direction on the two architectures examined.

To validate this result, we also measured the performance of a scenario that simply performs the `testAndMark()` operation on each object in the replay buffer (Figure 3.3). These results are shown in Table 3.6 and are normalized to the entire *mark* scenario. As

	Header		Side	
	Sync	Unsync	Sync	Unsync
Traverse	0.02	0.03	0.02	0.03
Enq-deq	0.10	0.10	0.10	0.10
Touch	0.15	0.15	0.15	0.15
Scan	0.40	0.40	0.39	0.40
Trace	0.59	0.59	0.58	0.59
Mark	**1.00**	0.84	0.98	0.83

(a) Pentium-M

	Header		Side	
	Sync	Unsync	Sync	Unsync
Traverse	0.02	0.02	0.02	0.02
Enq-deq	0.11	0.11	0.11	0.11
Touch	0.14	0.14	0.14	0.14
Scan	0.46	0.46	0.46	0.45
Trace	0.63	0.63	0.64	0.63
Mark	**1.00**	0.89	1.00	0.88

(b) AMD Athlon64

Table 3.5: Elapsed Time for Various Scenarios for Four Design Points, Normalized to the Synchronized Mark Scenario.

```
1  for item in buffer
2    obj = item.load()
3    obj.testAndMark()
```

Figure 3.3: Replay Scenario for Evaluating Mark-state Implementations

expected, this shows significant differences between the header and the side bitmap implementation, and as expected, both L1 and L2 cache miss rates decrease markedly with a side bitmap.

This result seems to confirm the intuition that a side bitmap provides a real locality advantage, but it contradicts the results in Table 3.5. We suspect that the discrepancy can be accounted for by the cache-displacing properties of the remainder of the tracing loop. The basis for the locality argument is that when marking an object, the required cache line will already be in cache with some probability which increases greatly as the metadata is densely packed into cache lines (spatial locality is *greatly* amplified by the dense bitmap). This argument depends on subsequent marks to the same line being relatively near to each other *temporally*, which is somewhat true when the mark is considered in isolation. However, when the mark is examined in the context of the entire tracing loop, subsequent marks to the same cache line will on average be much further apart in terms of memory accesses due to the significant memory activity of the remainder of the tracing loop (particularly due to the scan). This analysis suggests that any locality advantage due to dense metadata is almost entirely lost due to Am-

	Header		Side	
	Sync	Unsync	Sync	Unsync
Time	0.47	0.35	0.37	0.25
L1 Misses	0.50	0.50	0.36	0.36
L2 Misses	0.53	0.58	0.29	0.32
RI	0.27	0.25	0.29	0.27

(a) Pentium-M

	Header		Side	
	Sync	Unsync	Sync	Unsync
Time	0.46	0.27	0.38	0.22
L1 Misses	0.52	0.53	0.40	0.40
L2 Misses	0.54	0.54	0.41	0.41
RI	0.27	0.25	0.28	0.26

(b) AMD Athlon64

Table 3.6: Cost of The Mark Mechanism Alone for Four Design Points, Each Normalized to Cost of Entire Mark Scenario.

dahl's law and the predominance of memory activity in the scan portion of the trace loop, explaining the results in Table 3.5.

3.3.3 Heap Traversal Order

Another implementation decision that can have significant impact on the performance of a garbage collector is the choice of data structure for the work list, and the heap traversal order this induces. While it is 'common knowledge' among implementers that depth-first search outperforms breadth-first search, replay tracing gives us a platform for isolating the performance effect of traversal order *independent of the queue structure used*. We also look beyond this, to perform a thorough analysis of several other traversal orders that have not been studied in the literature. Research into parallel garbage collection algorithms in particular focus around work list designs that fairly share the collection work among the available collection threads, but we know of no work that evaluates the resulting performance in isolation.

3.3.3.1 Experiments

In these experiments, the scenario (in terms of the replay tracing framework) is the same, the full *mark* scenario from Section 3.3.1. The only difference between the data points presented below is the order in which objects are loaded into (and therefore replayed from) the replay buffer.

The data structures studied in this section are:

Sorted As a limit study, we include results for heap traversals in ascending address order, by sorting the replay buffer. This doesn't correspond to any realistic work-list data structure, but provides an interesting edge case. All graphs are normalised to this.

Stack A stack is the traditional data structure for the work list, and using it results in a depth-first traversal of the heap graph. This has been shown [Wilson et al., 1991] to produce good mutator locality when used with a copying collector, so it is reasonable to expect the same will follow for collector locality.

Queue A first-in first-out queue leads to a breadth-first traversal, with poor mutator locality when used in a copying collector.

Work Packet Ossia et al. [2002] describe a data structure in which each collector thread works with two packets of pointers, one as input to the traversal and one as output. When a thread exhausts its input packet, a new input packet is acquired from a global pool, and full output packets are flushed to the global pool. This technique maximizes the potential for work sharing among collector threads but also leads to a more breadth-first traversal order.

If the global pool is managed as a stack, this becomes a hybrid of the two techniques above, locally breadth-first, but globally depth first. We refer to this in the results as *work packet*. The traversal order changes as the size of the packets change, and we study the effect of the packet size on GC time. Hallberg [2003] indicates that the JRockit virtual machine uses this type of data structure, with a packet size of 493 items.

Buffered stack The prefetch work of Cher et al. [2004] relies on introducing a small FIFO buffer between the collector and the mark stack. Pointers popped from the top of the stack are inserted at the tail of the FIFO buffer, and the oldest entry in the buffer is processed. New pointers obtained from scanning the current object are pushed directly onto the stack.

This data structure is described in detail in Chapter 4 where our prefetch solution is described, but for now we analyse the locality effects of the ordering this produces. We look at the performance across a range of buffer sizes—a small buffer should be almost identical to a depth-first traversal, and as the buffer becomes larger we expect the order to start to resemble breadth-first.

Orthogonal to the actual traversal order, we also look at these configurations in an *edge enqueuing* collector. Edge enqueuing is discussed in detail in Section 4.5, but we introduce it briefly here so as to co-locate our performance results. Edge enqueuing differs from the standard (*node enqueuing*) in that when an object is scanned, its non-null pointer fields are pushed onto the work list instead of being marked and conditionally scanned. When an object is popped from the queue, it is marked, and only scanned if it wasn't already marked. Figure 3.4 shows the resulting loop structure.

Another issue in the traversal order of the heap (which is touched on later in Chapter 5) is the order of scanning of fields within an object. The 'natural' order is the order in which the reference fields are declared in an object's class and superclasses.

```
1  for p in root-set
2    queue.add(p)
3
4  while !queue.isEmpty()
5    obj = queue.remove()
6    if obj.testAndMark()   // Test mark-bit first
7      gcmap = obj.getGcMap()
8      for p in gcmap.pointers()
9        child = p.load()
10       if child != null
11         queue.add(child) // Enqueue without testing mark-bit
```

Figure 3.4: The *Edge-Enqueuing* Tracing Loop

3.3.3.2 Results

This section shows results of our experiments exploring traversal order. We measure total elapsed time for the heap traversal across all GCs in the third iteration of each benchmark, and repeat each measurement six times in order to obtain a sufficiently small confidence interval. We take the arithmetic mean of these six iterations and divide by the mean elapsed time for the 'sorted' traversal order to produce a normalised elapsed time ratio for each benchmark and traversal order. We summarise this to a single figure for the traversal order on a machine architecture by taking the geometric mean of the per-benchmark normalised figures.

Platform	Clock	DRAM	L1 D	L1 I	LLC
Core i7 920	2.6GHz	4GB	32KB	32KB	8MB
AMD Phenom II X6 1055T	2.8GHz	4GB	64KB	64KB	6MB
Core i5 670	3.4GHz	4GB	64KB	64KB	4MB
Core 2 Duo E7600	3.1GHz	4GB	32KB	32KB	3MB
Atom D510	1.8GHz	4GB	32KB	32KB	1MB

Table 3.7: Hardware platforms used for heap traversal order experiments.

We perform these experiments on the five architectures listed in Table 3.7. This differs from the platforms used for the above experiments, because these experiments were performed at a later time and the machines used in the above experiments were no longer available.

Depth-first versus breadth-first Figure 3.5 shows the results of a comparison between a Stack, a Queue and a Work Packet structure with mid-size packets of 128 entries. As expected, a stack performs much better than a queue, showing that the locality effects of a depth-first search order apply to the collector as well as the mutator. In fact the stack performs surprisingly well compared to the address-order (sorted) traversal to which the graph is normalised. Also of interest is how poorly a breadth-first search performs, taking over 50% longer than depth-first search on all architec-

tures. Unsurprisingly the work packet ordering performs somewhere between the two.

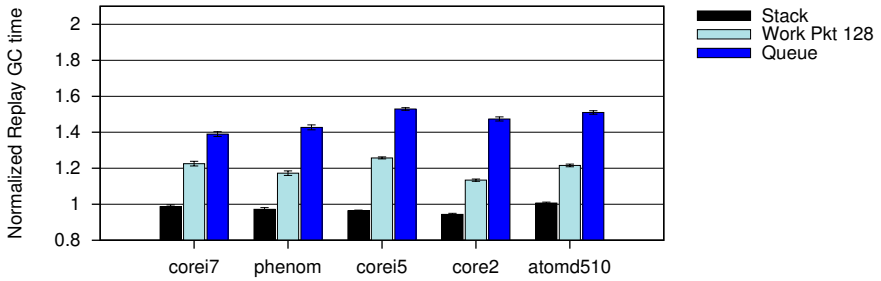


Figure 3.5: Effects of traversal order: major design choices

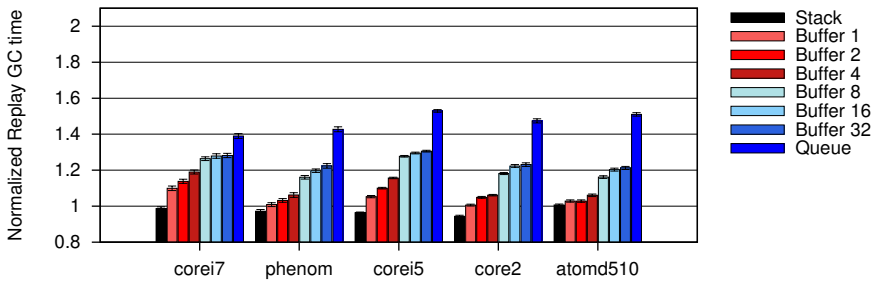


Figure 3.6: Effects of a FIFO buffer

FIFO buffer Figure 3.6 shows the impact of adding a Cher et al. style FIFO buffer to an otherwise depth-first traversal, for varying buffer sizes. The data series 'buffer n' shows the impact of a buffer of size n, varying from 1 element to 32 elements. For comparison, we also include the data point for breadth-first search.

The impact of the FIFO buffer is surprisingly severe. A 32 element FIFO buffer slows down this scenario by almost 30% over a depth-first search, and on the Core i7 is almost as significant as a breadth-first search. Using this data structure without prefetch is clearly not helpful.

Work Packet Figure 3.7 shows the effect of different size packets in the work packet queue's partial breadth-first order. Perhaps unsurprisingly given the results in Figure 3.6, even an 8-entry work packet has a large impact on replay time. Interestingly the size of the packet seems to have little consistent effect. As with the FIFO buffer, the slowdown is never quite as pronounced as a full breadth-first ordering.

Scan Direction Figure 3.8 compares scanning through each object's fields either forwards or backwards. The impact (given a depth first traversal) is significant. This

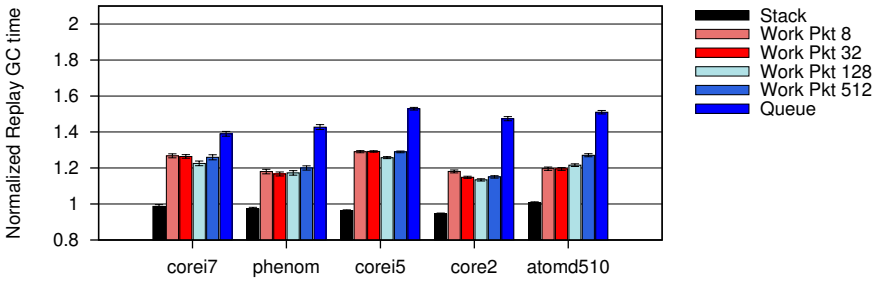


Figure 3.7: Effects of partial breadth-first (work packet) traversal order

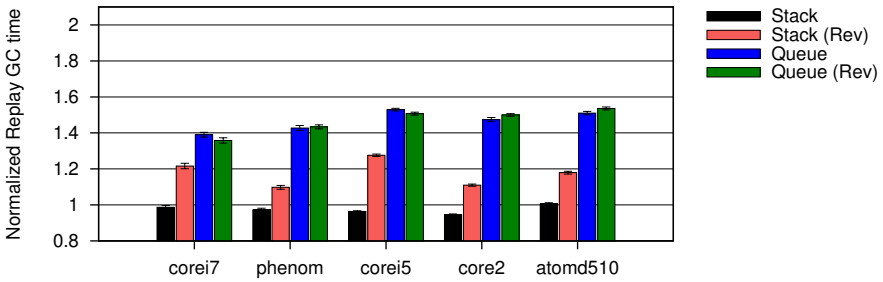


Figure 3.8: Reversing the order of scanning of fields

tallies with Gu et al. [2006], who noticed consistent slowdowns when scanning objects in reverse order. Interestingly when used in a breadth-first search, the order of scanning within the object makes little difference—evidently it is difficult to make a bad traversal order worse.

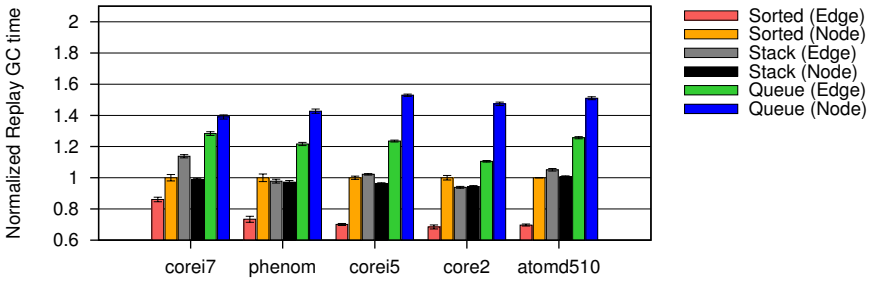


Figure 3.9: Edge enqueuing.

Edge Enqueuing Figure 3.9 shows results for the main traversal orders in an edge-enqueuing collector, normalised like the graphs above to the performance of a node-order sorted traversal. As expected, the best-case performance of edge enqueuing, when the heap is traversed in address order, is much better than node-enqueuing. The

difference is least pronounced on the Core i7 processor, where a significant percentage of the benchmarks fit completely in its 8MB L3 cache, and increases as we move to the right across the graph, and the last-level cache size decreases. This confirms that main-memory latency effects are responsible for the difference in performance.

Under a stack discipline, the improvement disappears, and performance of the edge-enqueued collector is slightly slower that of the node enqueued collector. This is reasonable since edge-enqueuing performs `testAndMark` on the same number of objects, scans the same number of objects, but enqueues more objects than node-enqueuing. In the replay tracing harness, where enqueuing is a no-op, this is the expected result.

Under a queue discipline, edge enqueuing performs significantly better than its node-order counterpart, a difference that increases as cache size drops. One plausible explanation is that while breadth-first traversal causes poor cache-miss behaviour on the initial fetch of an object's header, the additional heap references in the node-enqueuing loop's kernel add a second source of cache misses to objects that have been evicted from cache by the poor traversal order. The edge-enqueued loop only suffers the initial cache miss.

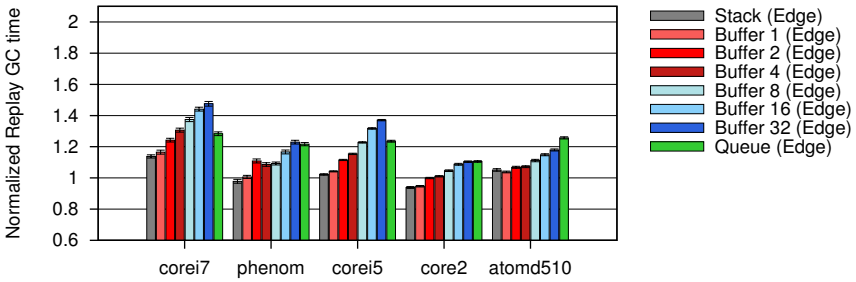


Figure 3.10: Edge enqueuing with a FIFO buffer

Figure 3.10 shows that a FIFO buffer has a similarly detrimental effect as on the node-enqueued traversal. Interestingly though, particularly on the 'Core' processors, the buffer leads to worse performance than a breadth-first search.

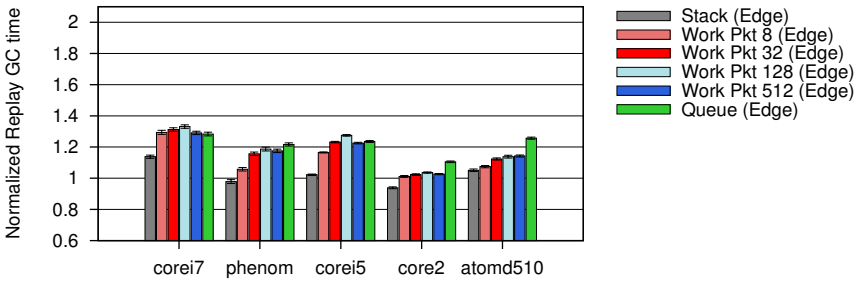


Figure 3.11: Edge enqueuing. Effects of partial breadth-first order

Results for the locally breadth-first (work packet) order are shown in Figure 3.11. In an edge enqueueing collector, this ordering performs similarly to its node-enqueued counterpart. The only significant difference is that because of the better performance of breadth-first with edge enqueuing, breadth-first performs better than any of the locally breadth-first orderings.

3.3.3.3 Conclusion

Heap traversal order has a significant effect on the performance of a garbage collector, and interestingly the depth-first ordering resulting from using a stack for a work list is very close to optimal. While the magnitude of the difference we demonstrate using replay tracing is larger than that which would be experienced in a practical system, these results demonstrate that traversal order must be a first order consideration in the design of the tracing loop.

3.4 Summary

This chapter presented a new approach to analysing the performance of a garbage collector. We used this technique to a) identify the costs of various components of the main loop of the Garbage Collector, b) evaluate two implementations of object mark state, and c) look at the impact of heap traversal order on the garbage collector, independent of the performance of the data structure used for the work list. This information is valuable when targeting optimizations of the garbage collector, and is critical in motivating the work in the remainder of this thesis.

Chapter 4 uses the analysis from Section 3.3.1 to design a software prefetch strategy that provides consistent speedups across a wide range of benchmarks. Following that, Chapter 5 looks at the scanning mechanism, the next most expensive component of the tracing loop.

Chapter 4

Effective Software Prefetch

The previous chapter presents an analysis of the costs of the mark operation in a mark-sweep garbage collector. In this chapter we build on insights gained from this analysis. We show how *edge order* traversal of the heap improves locality and allows us to speed up the mark loop through software prefetch. In order to take best advantage of the locality gains from the edge-order traversal, we also present a new implementation technique for object mark state that stores this metadata in object headers without sacrificing speed in the sweep operation.

This chapter is based on work published in the paper "Effective Prefetch for Mark-sweep Garbage Collection" [Garner, Blackburn, and Frampton, 2007]. The key contribution of this work is the cyclic mark-state design described in Section 4.3, and the combination of edge-order heap traversal with buffered prefetch. The mark-state design was contributed to the Jikes RVM codebase in 2006, and remains the basis for its MarkSweep collector.

4.1 Introduction

Many garbage collection algorithms perform a transitive closure over the heap identifying live objects without moving the objects. The best known of these is the mark-sweep collector, but mark-compact, Immix and others all have similar closure operations. The performance of this closure operation is dominated by cache miss time, because from a locality perspective it is a random walk across the entire heap. This renders caches in modern processors essentially ineffective, and similarly defeats hardware prefetch mechanisms which are built to identify and predict regular patterns of memory accesses. Previous attempts to apply software prefetch to this problem have produced little or no speedup.

In modern computer systems, access latencies—the time between issuing a memory access instruction and the arrival of the data in the CPU—increase significantly through the cache hierarchy. Level 1 cache has a typical latency of 1–4 clock cycles, whereas main memory can have a latency of over 100 cycles.

Broadly, there are two main techniques that can be used to improve memory behavior of software. The first is to minimise memory accesses by combining spatial and temporal locality of data operations, that is to keep data that is accessed closely

in time closely in space. This can be done by rearranging data structures (e.g. column-major versus row-major array layouts), or by changing data representation to be more spatially efficient (e.g. a bit-map versus a byte-map). The improvement is achieved by making more efficient use of finite resources such as caches, TLBs etc.

The second technique is *prefetch*, which attempts to mitigate main-memory access latency by requesting data in advance of actually using it, moving it into cache so that it is available quickly when the program actually requires it. Prefetch can be implemented in either hardware or software.

Hardware prefetch units observe the access patterns of the running program and attempt to predict future accesses with no intervention by the programmer or runtime. This is very successful for applications such as scientific computations involving vector or matrix algebra, but are not amenable to the irregular patterns of *pointer chasing*, such as those that dominate a garbage collection trace.

Software prefetch requires additional instructions to be inserted into the program so that access patterns that hardware prefetch cannot predict can be exploited. The potential benefits of this approach are limited by the ability to predict accurately which memory will be required in the near future.

4.2 Related work

The basic mark-sweep algorithm has been continuously refined since its initial introduction in 1960 [McCarthy, 1960]. Our work builds directly upon prior work on lazy sweeping [Hughes, 1982; Boehm, 2000] and prefetching [Boehm, 2000; Cher et al., 2004].

Boehm [2000] evaluates lazy sweeping in the context of the Boehm-Demers-Weiser (BDW) collector. Instead of sweeping the whole heap immediately after each mark phase, the GC only sweeps completely free (unmarked) blocks of memory, and the remainder are lazily swept on demand by the allocator. This approach is effective when mark bits are maintained on the side, allowing free blocks to be cheaply identified. Boehm saw up to 17% net performance win from lazy sweeping.

In the same paper, Boehm is the first to apply software prefetching to garbage collection. He introduces a prefetch strategy called *prefetch on gray*, where an object is prefetched upon insertion into the mark stack. This strategy is somewhat effective in a heap implementation using a mark bitmap on the side. Boehm saw speedups of up to 17% in synthetic GC-dominated benchmarks and 8% on a real application (ghostscript). However, when mark bit metadata is embedded in the object header, objects are guaranteed to be in cache when they are pushed on the mark stack, obviating the need for a prefetch. Prefetch on gray has two key limitations. Many items are prefetched too soon, and by the time the depth-first search pops the stack back to the item, it has been evicted from cache by the intervening memory activity. Secondly, many items are prefetched too late, because the last object pointed to by any given object is accessed immediately after it is pushed on the stack and prefetched, allow-

ing no time for the prefetch to take effect. Boehm measures costs using profiling, and reports results for a small number of C benchmarks.

```
1  void add(Address item) {
2    stack.push(item)
3  }
4
5  Address remove() {
6    Address pf = stack.pop()
7    pf.prefetch()
8    fifo.insert(pf)   // FIFO buffer allows
9    return fifo.pop() // prefetch time
10 }
```

Figure 4.1: The FIFO-Buffered Prefetch Queue [Cher et al., 2004].

Cher et al. [2004] build on Boehm's investigation, using simulation to measure costs and explore the effects of prefetching in the BDW collector. They find that when evaluated across a broad range of benchmarks, Boehm's prefetch on gray strategy attains only limited speedups under simulation, and no noticeable speedups on contemporary hardware. Cher et al. introduce the buffered prefetch strategy that we also adopt (see Figure 4.1). Buffered prefetching observes that the LIFO discipline used in the mark stack when performing depth-first search is unsuitable for prefetching because the pattern of future accesses in a LIFO structure is hard to predict. They recover predictability by placing a small FIFO prefetch buffer between the mark stack and the tracing process. When the tracing loop pops the next entry from the mark stack (line 6), a prefetch is issued on its referent (line 7) and the entry is inserted at the tail of the prefetch FIFO (line 8). The entry at the head of the prefetch FIFO is then selected for scanning (line 9). The depth of the FIFO defines the prefetch distance.

Cher et al. validate their simulated results using a PowerPC 970 (G5), almost identical to the PowerPC system on which we obtain our results. They obtain significant speedups on benchmarks from the jolden suite, but less impressive results for the SPECjvm98 suite, with their best result being 8% on _202_jess, and 2% on _213_javac. All of these results were achieved in very space-constrained heaps; about 1.125× the minimum heap size, sufficiently small that GC time is a large fraction of total time, thereby amplifying the effect of any GC improvements. In Section 4.4 we show that by combining Cher et al.'s approach with *edge ordered enqueuing*, we see consistent, sizable performance improvements across a large number of benchmarks.

Groningen [2004] uses a similar prefetch buffer to Cher et al. to speed up mark-sweep collection for the functional language Clean. They evaluate on several machines including an AMD Opteron of comparable speed to our Athlon 64, and achieve an increase in throughput of up to 40%. The differences in language and benchmarks make this result difficult to compare to Cher et al. or our work.

Hallberg [2003] applies prefetch to the JRockit virtual machine, and evaluates her implementation on an Itanium processor using SPECjbb2000. JRockit uses a 'work packet' work queue, that we show in Section 3.3.3.2 to be significantly slower than

a stack. It is difficult to evaluate this work against ours, because of the difference in hardware platforms and the queue discipline.

4.3 Key Mark-Sweep Design Choices

Although the basic mark-sweep algorithm is well documented and well understood, there are several design choices that have the potential to significantly affect performance [Wilson et al., 1995; Jones and Lins, 1996; Boehm, 2000]. These include: a) the allocation mechanism and free-list structure, b) the mechanism for storing mark state, c) the technique used to sweep the heap, and d) the structure of the collector's work queue.

4.3.1 Allocation

We use MMTk's standard allocation mechanism and free-list structure for all mark-sweep configurations we evaluate in this thesis. The free list is structured as a *two level segregated fit* free-list structure [Wilson et al., 1995]. The allocator divides memory into coarse grain *blocks* of which there are several distinct sizes ranging from 4K to 32K bytes. When required, the allocator assigns individual blocks to a single *object size class* and divides them into N equal sized cells. The allocator always satisfies requests with the first entry on the free list for the smallest size class that is large enough. This approach reduces worst case fragmentation while ensuring a fast simple allocation path because any request is always satisfied by the head of the free-list, and the choice of free list is generally statically determined by the compiler.

4.3.2 Mark state

As discussed in Section 3.3.2, the collector's mark phase depends on being able to store mark state for each object. We examined the performance of each of these options, and discovered that while the side bitmap appears to have a locality advantage, in a real collector these advantages disappear due to the cache flushing effects of the collector. We look at both of these design choices when evaluating our prefetch optimizations.

An important optimization for *header* state is to change the sense of the mark bit at each garbage collection, which avoids having to reset all mark bits after every collection. Side bitmaps can be trivially and cheaply zeroed in bulk, thereby avoiding this issue.

4.3.3 Sweep

A classic mark-sweep collector will sweep the entire heap at the end of each collection, identifying unmarked memory and returning it to free lists. The sweep comprises two components: examining each block's metadata to identify marked objects, and populating the free lists with any freed memory. Scanning a side bitmap will easily reveal blocks which are entirely free, allowing them to be freed up entirely, therefore

avoiding the construction of a free list in these blocks. Sweeping the entire heap at each collection is known as *eager sweeping*.

Boehm [2000] noted significant advantages to *lazy sweeping*. A lazy sweeper sweeps blocks only when they are required by the allocator. This has two significant advantages. First, it saves sweeping work for many blocks which are unchanged from collection to collection. Second, free list construction occurs immediately prior to use of the cells, which has measurable temporal locality benefits. Blocks which are entirely free can be identified by examining a side bitmap at the end of a collection. Lazy sweeping is therefore normally used with a side bitmap. Our design effort was focused on combining the performance gains of lazy sweeping with the locality benefits of header metadata we exploit in Section 4.5.

4.3.3.1 Block marks

In the process of teasing apart the various design choices for mark-sweep collectors, we wanted to explore lazy sweeping with header mark bits. As we show in Section 4.6 this combined approach is quite effective, although to our knowledge has not been explored before. In order to free unused blocks eagerly while lazily sweeping partially used blocks, we developed *hybrid marking*, which uses mark bits in object headers and a single byte of side metadata for each block. Each block's side metadata is set whenever an object within the block is marked. Any block with an unmarked metadata byte is completely free, and may be eagerly reclaimed at collection time. This way we are able to combine header metadata with lazy sweeping.

In performance terms, the per-block mark state makes an insignificant contribution to the mark phase, because it is an *unconditional write* operation. Multiple marks for the same block may be absorbed in the on-chip write buffer, and no pipeline stalls will occur as long as there is capacity in the write buffer. The space overhead of the block marks is also insignificant, requiring one byte per 4KB block.[1]

4.3.3.2 Cyclic mark state

Recall that one of the two advantages of lazy sweeping is that unmarked objects only lazily make their way onto free lists. When a side bitmap is used, this delay is of no consequence beyond the obvious saving of work. Recall however that header mark state changes sense at each collection. Therefore an object with header mark state will not be recognized as unmarked if it is swept an even number of collections since it was last marked. This situation can never lead to a live object being collected since every live object is visited at every collection, but it can lead to *floating garbage*. Of course this situation does not arise with a side bitmap since the side bitmap can be reset cheaply, and therefore all unused objects are guaranteed to be unmarked.

Floating garbage may become a problem when the allocation behaviour of a program varies over time. In a segregated free list allocator, blocks remain allocated to a

[1] As mentioned above, blocks vary in size. For simplicity (and to speed up the calculation of the metadata addresses) we reserve metadata for each 4KB chunk.

given size class until there are no live objects in the block, in which case it is returned to the pool of free blocks and can be re-used for a different size class. When a garbage collection is triggered, it is likely that there are unswept blocks in some size classes due to a relatively low demand for objects of that size class in the previous mutator phase. Eventually the mutator may again demand significant numbers of objects in these size classes, and blocks that have not been swept for one or more GC cycles will come to be swept.

We compared two basic solutions to the problem of floating garbage when lazy sweeping. One approach is to simply sweep the unswept portion of the heap at the beginning of each collection cycle. This approach will still see the locality advantages of lazy sweeping and successfully eliminate floating garbage, but loses the advantage of avoiding redundant sweeping and in practice it performs badly. Our second approach uses multiple bits to record mark state within each object header and stores the mark state as the collection number modulo the largest number that can be stored in the header bit field. With a bit field size of n, we reduce the worst case amount of floating data by a factor of 2^{-n}. Note that data will only float in the case that it is finally swept by the mutator exactly $N \times 2^n$ collections since it was last marked without that block being swept in any of the previous 2^{n-1} collections. In practice we find that only 4 bits of cyclic mark state are required before the performance gains of additional bits become insignificant.[2]

4.3.4 Work queue

Section 3.2 described the basic tracing loop (Figure 3.1), and pointed out that the collector's work queue could be maintained in either LIFO or FIFO disciplines, leading to depth-first or bread-first traversals of the heap respectively. In a copying collector the traversal order affects the relative position of objects after collection, and therefore impacts application locality [Huang et al., 2004; Wilson et al., 1991]. In such collectors, depth first orders are generally accepted as more efficient [Huang et al., 2004; Wilson et al., 1991]. In Section 3.3.3 we show that traversal order is also significant for collection speed, so we use a LIFO (stack) for the experiments in this chapter.

4.4 Software Prefetching

The results presented in Section 3.3.1 can now be seen in context. The initial access of an object when it is fetched from the work queue, and the subsequent scanning accounts for approximately 40% of the scan time. The subsequent references to the referents of the scanned object account for another 40% of the time. In order to maximise its effectiveness software prefetch needs to address *both* sources of cache misses.

These results demonstrate that poor locality is the principal bottleneck in the performance of the tracing loop. Because of the significant miss penalties imposed by

[2]This is the default implementation in MMTk since Jikes RVM release 2.5.

modern architectures, it is clear that improving the cache behavior of the trace will be fruitful in terms of improving overall trace performance.

Successful software prefetch depends on two factors. Firstly, we need to accurately predict memory accesses, because prefetching memory that we subsequently do not access increases cache pressure and damages performance. Secondly we need to judge the correct *prefetch distance*, that is the time between issuing the prefetch and accessing the prefetched memory. If our prefetch distance is too small, the processor may stall waiting for data to arrive in cache. If it is too large, the data may be evicted from cache before we access it, resulting in a second load from main memory and causing even more memory traffic.

4.4.1 Prefetching For GC Tracing

Previous work [Boehm, 2000; Cher et al., 2004] has focused on the potential for prefetching objects from the marking stack with a tracing loop similar to the one in Figure 3.1. Ignoring memory accesses directly associated with the queuing mechanism (which we have shown in Section 3.3.1 to be insignificant), the two sources of memory access are a) *marking* each object (line 11), and b) *scanning* each marked object (lines 7, 8 and 9). The first requires the object's mark metadata, which may be in the object's header, or held on the side. The second requires accessing the object's GCmap (normally in the object's header or accessed via indirection from the object's header), and scanning each of the pointers within the object. The performance of the scanning mechanism is the subject of Chapter 5.

We take as a starting point the prefetch buffer used by Cher et al.

4.5 Edge Order Traversal

The standard tracing loop is depicted in Figure 4.2a and is designed to minimise the number of objects that need to be enqueued and dequeued from the marking stack. Other alternative tracing loops are possible [Jones and Lins, 1996] but not generally used due to the additional stack operations they entail. However, as the analysis in the previous section has shown, *queuing operations are not the bottleneck to improving performance*. We observe that it is possible to remove one of the memory access points in the loop by enqueuing *all* non-null objects during the *trace* rather than inspecting these objects and filtering out marked objects. Figure 4.2b gives pseudo-code for this approach, which we call *edge* enqueuing, because the referents of all non-null edges in the graph are placed on the stack. We refer to the traditional technique as *node* enqueuing (since each node in the graph is enqueued only once). Comparing Figures 4.2a and (b), the only difference between the two loops is that the mark operation (line 13) of Figure 4.2a) is hoisted to line 8 in Figure 4.2b.

By hoisting the mark operation, edge enqueuing weakens the guard on line 14 of Figures 4.2a and (b), so that children are eagerly enqueued, and the conditional mark operation is only performed later, immediately before the child is scanned. This increases the number of queue operations from the number of *nodes* in the live object

```
1  for p in root-set
2    obj = p.load()
3    mark(obj)
4    queue.add(obj)
5
6  while !queue.isEmpty()
7    obj = queue.remove()
8
9    gcmap = obj.getGcMap()
10   for p in gcmap.pointers()
11     child = p.load()
12     if child != null
13       if child.testAndMark()
14         queue.add(child)
```

(a) The Standard Tracing Loop

```
1  for p in root-set
2
3
4    queue.add(p)
5
6  while !queue.isEmpty()
7    obj = queue.remove()
8    if obj.testAndMark()  // hoist from 13
9      gcmap = obj.getGcMap()
10     for p in gcmap.pointers()
11       child = p.load()
12       if child != null // weaker guard
13
14         queue.add(child)
```

(b) The Edge-Enqueuing Tracing Loop

Figure 4.2: Comparing the Standard and Edge Enqueuing Tracing Loops

graph to the number of *edges* in the live object graph, but does not affect the number of objects which are marked or scanned. As we have already shown, queuing operations form a negligible part of the cost of tracing. The benefit of edge enqueuing is that the *mark* (line 8 of Figure 4.2b), *scan* (lines 9–11) and *trace* (lines 13 and 14) for a given object now occur contemporaneously, providing a far more predictable access pattern, which is more amenable to prefetching. The additional mark stack space requirements of edge enqueuing are also reasonable. We found that the SPEC and DaCapo benchmarks have on average 40% more edges than nodes, leading to about 40% more queue operations.

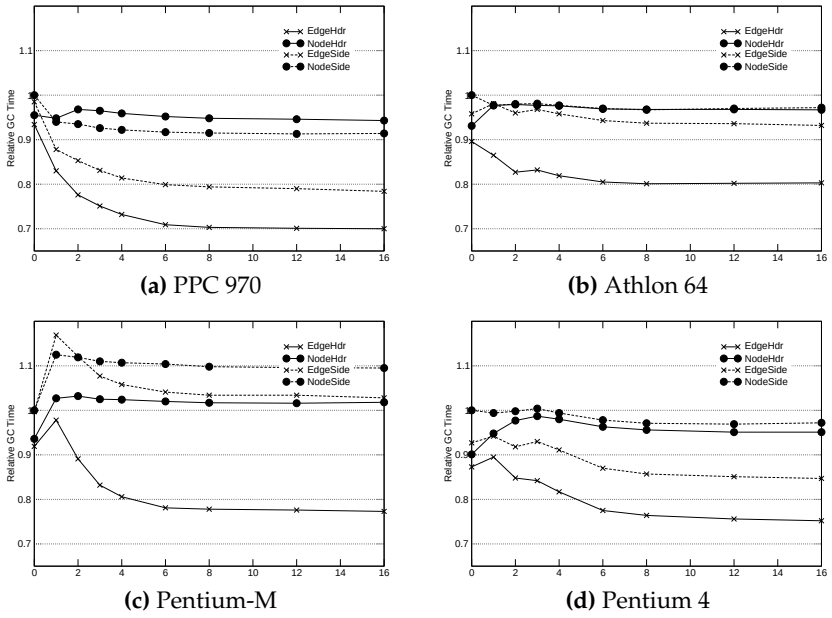
The contemporaneous *mark*, *scan* and *trace* mean that edge enqueuing has better temporal locality than node enqueuing. This addresses the first of our identified techniques for improving locality, and also provides a better environment for prefetching, since all accesses to each object occur together when that object is removed from the queue (in line 7). If the mark state is kept in the object header rather than a side data structure, we also achieve better *spatial locality*, and for the majority of objects all operations will take place on one or two cache lines. We implemented the FIFO-buffered mark queue (Figure 4.1 [Cher et al., 2004]) in our infrastructure to explore the effect of node and edge enqueuing on prefetching. We control the prefetch distance by changing the size of the FIFO buffer.

4.6 Performance Results

We now evaluate the effectiveness of software prefetching in the tracing loop under both *edge* and *node* enqueuing models. Given their effect on locality, we compare with both *header* and *side bitmap* implementations of mark state, yielding four configurations.

Platform	Clock	DRAM	L1 cache	L2 cache
AMD Athlon 64 3500+	2.2GHz	400MHz DDR2	64KB 64B 2-way	512KB 64B 16-way
Pentium-4 *Prescott*	3.0GHz	533MHz DDR2	16KB 64B 8-way	1MB 64B 16-way
Pentium-M *Dothan*	2.0GHz	533MHz DDR2	32KB 64B 4-way	2MB 128B 8-way
PowerPC 970 *G5*	1.6GHz	333MHz DDR	32KB 128B 2-way	512KB 128B 8-way

Table 4.1: Hardware Platforms for prefetch experiments



(a) PPC 970 **(b)** Athlon 64

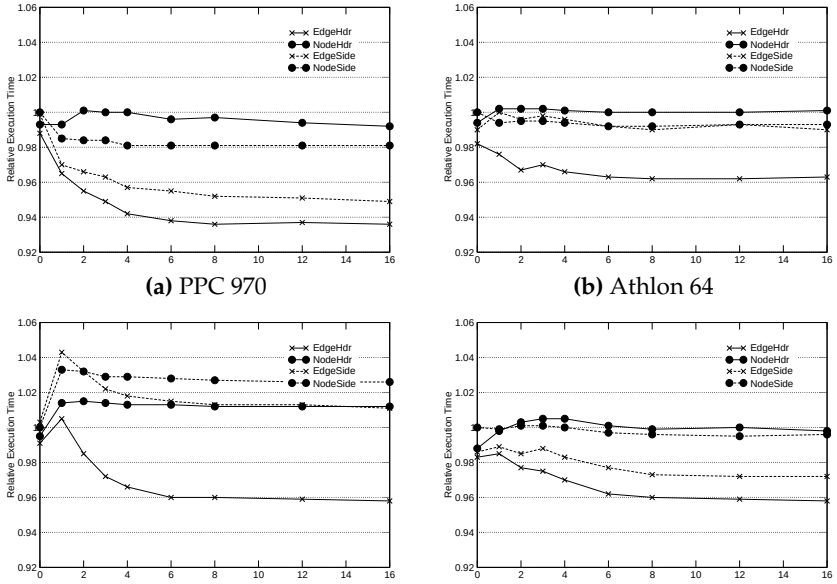
(c) Pentium-M **(d)** Pentium 4

Figure 4.3: Normalized GC Time vs. Prefetch Distance, as Geometric Mean of 17 benchmarks. Four Combinations of Edge and Node Enqueuing and Side and Header Metadata Are Shown.

We evaluate performance on 4 architectures, details of which are provided in Table 4.1.

A side bitmap incurs the overhead of a synchronized update to ensure correctness for parallel collection (which we measured in Section 3.3.2). Since the synchronization does not change the overriding memory access patterns, the synchronization overhead should be independent of the presence of prefetch, and cause only a minor shift with respect to the prefetch distance due to the additional cycles required during processing. Our intuition is that the byte-map approach—while avoiding a synchronized operation on mark—would have worse locality properties than that of a dense bitmap by consuming eight times more cache lines.

For this loop, it is natural to measure prefetch distance in terms of loop iterations, and hence numbers of elements in the prefetch buffer. Figures 4.3 and 4.4 show garbage collection and total time respectively, as we vary the prefetch distance from 0 to 16 for each of the four configurations (EdgeHdr, NodeHdr, EdgeSide, NodeSide), mea-



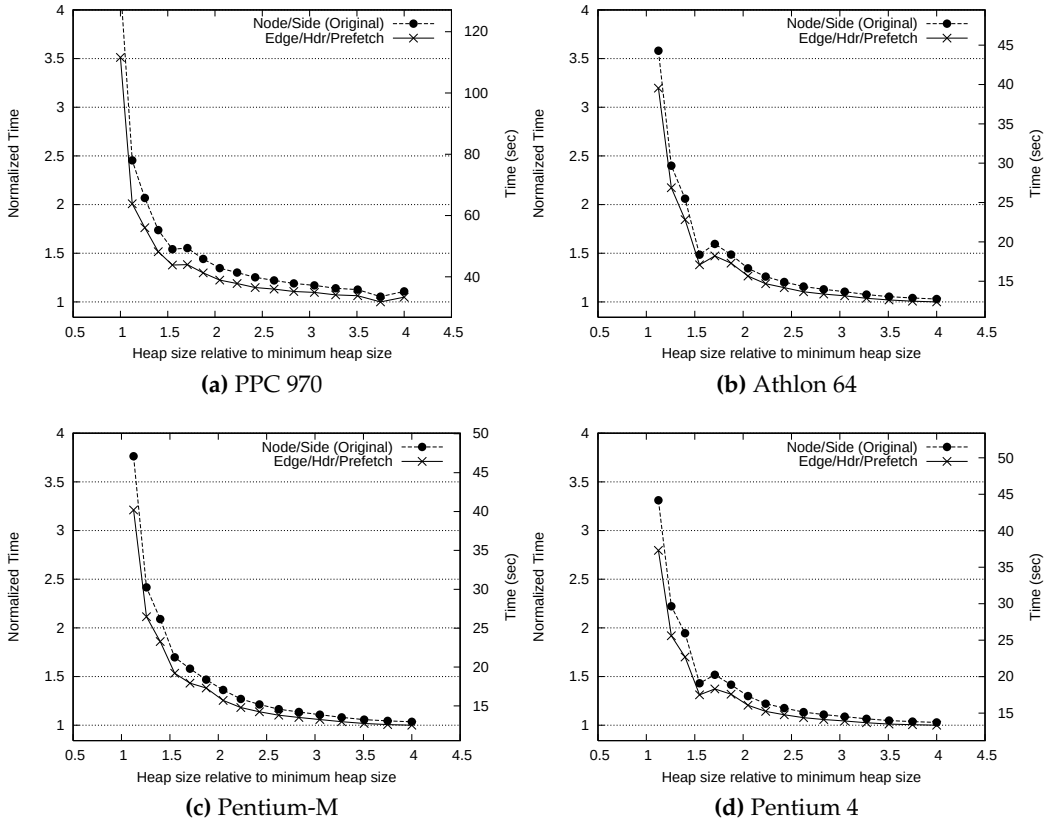
(a) PPC 970 **(b)** Athlon 64

(c) Pentium-M **(d)** Pentium 4

Figure 4.4: Normalized Total Time vs. Prefetch Distance, as Geometric Mean of 17 benchmarks. Four Combinations of Edge and Node Enqueuing and Side and Header Metadata Are Shown. Measured In a Generous Heap (3 × Minimum).

sured on our four modern architectures. Each graph shows the geometric mean of performance for the full set of 17 benchmarks drawn from DaCapo and SPEC. Results are normalized to the time for NodeSide with no prefetch. NodeSide is the configuration which most closely follows prior work [Boehm, 2000; Cher et al., 2004]. We gathered results for different heap sizes, but found that the effect of prefetch was independent of heap size. We report here the results for a fairly generous heap; three times the minimum heap size for each benchmark. We performed identical experiments on 2× and 4× heaps, and found the GC-time results were almost indistinguishable, although the overall impact of this GC-time optimization on total time obviously increases since time spent in GC goes up in smaller heaps. For completeness, Figure 4.5 shows total time as a function of heap size. In order to maintain a consistent number of collections across the different configurations and fairly assess the effect of prefetch, we did not allow the header meta-data configurations to make use of the minor space saving due to avoiding the side bitmap.

The results in Figure 4.3 show a clear win for EdgeHdr (edge order enqueuing with mark state in the header), which out performs all other configurations on all four architectures. The impact of prefetching ranges from modest to poor for node order enqueuing. This poor result for node enqueuing is consistent with prior work which saw only modest improvements with prefetching [Cher et al., 2004; Boehm, 2000]. The difference in effectiveness of the prefetch operations across the architectures is



(a) PPC 970 **(b)** Athlon 64

(c) Pentium-M **(d)** Pentium 4

Figure 4.5: Relative Total Execution Time As a Function of Heap Size, Comparing Node Order and Side Mark Against Prefetching With Edge Order and Header Mark. Geometric Mean of 17 benchmarks.

significant, with GC-time performance improvements ranging from 30% on the PowerPC 970, to around 20% on the Athlon 64. Notable degradations occur for a number of configurations on the Pentium-M. We also see that prefetch distances greater than eight provide little advantage, with longer prefetch distances degrading performance slightly on the Athlon 64.

In order to perform a direct comparison with Cher et al., we followed their approach and measured the speedup in total time in a very space-constrained heap, 1.125× the minimum heap (left-most data in graphs in Figure 4.5). In this situation, we obtain total running time speedups of 18%, 15%, 11% and 16% on the PowerPC 970, Pentium-M, AMD and Pentium 4 respectively. This compares very favorably with Cher et al.'s result of 6% speedup on the PowerPC 970 (achieved with arguably more amenable benchmarks), and is consistent with our results which show that edge enqueuing is essential to effective software prefetch.

4.7 Robustness: Experiences With Other Code Bases

Since submitting this work for publication in 2006, we ported the work to a substantially different version of the Jikes RVM code base. Our efforts to reproduce our original results were frustrating, but ultimately illuminating. Our experience may be relevant to anyone wishing to implement prefetching in their own garbage collector.

First, poor code quality in the compiled code for the tracing loop can dominate any prefetching advantage. Through painstaking detective work we established that the absence of a number of aggressive optimizations resulting in the addition of an extra register spill reduced the tracing loop performance sufficiently that the prefetching advantage was negligible or zero. This was an unintended artifact of changes in the underlying Jikes RVM code base.

Second, the prefetch optimization only improves the tracing loop. In a garbage collector where the tracing loop does not dominate performance, the usefulness of this optimization will be correspondingly diminished. The way Jikes RVM collects its 'boot image' was changed from tracing (which utilizes the prefetch) to an explicit enumeration of pointer fields (which does not). We found that in small benchmarks where the Jikes RVM boot image formed a large fraction of the workload, the effectiveness of the prefetch optimization was significantly diminished.

Finally, we found that on some architectures the FIFO structure placed in front of the mark stack Cher et al. [2004] gave a performance advantage even *without* the prefetch. This is evident in Figure 4.3(c) where performance improves at x=1, and we saw similar behavior on an Intel Core 2 Duo. This is particularly surprising given the traversal order results in Section 3.3.3. Our best guess is that the FIFO structure enabled more aggressive hardware speculation through more predictable access patterns, just as the FIFO facilitates software prefetch.

4.8 Summary

This chapter presented a new approach to software prefetch for garbage collection that achieves significant, consistent speedups across a wide range of benchmarks and architectures. We also introduce a new mark state implementation that allows fast lazy sweeping with header mark state. By combining header mark-state information with edge enqueuing, we change the locality of the tracing loop making it amenable to buffered prefetch. The combination of all 3 is necessary to achieve this result.

The design of this approach was based on insights gained into the performance characteristics of the tracing loop using the replay tracing framework presented in Chapter 3. The next chapter looks at the second most costly factor identified by our analysis, that of the object scanning operation.

Chapter 5

Object Scanning

> *Perhaps the most important and pervasive principle of computer design is to focus on the common case: in making a design trade-off, favor the frequent case over the infrequent case. This principle applies when determining how to spend resources, since the impact of the improvement is higher if the occurrence is frequent.*
>
> Hennessy and Patterson [2006, p. 38]

Chapter 3 analysed the performance of the garbage collection tracing loop, and found as expected that two components dominated the cost of the loop. The first of these, the *mark* operation was analysed in Chapter 4, and we identified a way to use software prefetch to speed up this operation. This chapter looks at the second of the two major costs, scanning the object to find its pointer fields.

This chapter is based on work published in the paper 'A Comprehensive Survey of Object Scanning Techniques' [Garner, Blackburn, and Frampton, 2011]. The paper makes several contributions to the literature: a clear enumeration of the design space of object scanning techniques; a direct comparison of the performance of the most interesting points in this space, over several architectures and a large number of benchmarks; and a new design, using bits in the object header, that performs well on most benchmarks and is comparatively simple to implement. The best performing design from this study is now the default object scanning mechanism of Jikes RVM.

This chapter is structured as follows. Section 5.1 expands on the problem domain, outlines our analysis methodology and introduces the results. Section 5.2 surveys what little literature there is on this topic. Section 5.3 looks at the object demographics of the benchmarks we evaluate, providing statistical information we use to create some optimizations. Section 5.4 looks at the design space for scanning techniques, and informs our choice of configurations to evaluate. Section 5.5 details the performance evaluation methodology used to produce the performance results presented in Section 5.6.

5.1 Introduction

Enumerating object reference fields is key to all *precise* garbage collectors. For tracing collectors, liveness is established via a transitive closure from some set of roots. This requires the collector to identify and then follow all reference fields within every reachable object. For reference counting collectors, once an object's reference count falls to zero, each of its referents must be identified and have its reference count decremented. The process of reference field identification is known as *object scanning*. In order to be precise in the absence of hardware support, object scanning requires assistance from the language runtime. Otherwise, tracing must *conservatively* assume all fields are references [Boehm and Weiser, 1988; Boehm, 1993; Jones and Lins, 1996]. This chapter quantitatively explores the design tradeoffs for object scanning in precise garbage collectors.

Object scanning is performance-critical since it constitutes the backbone of the tracing mechanism, and therefore may be executed millions of times for each garbage collection. The extensive literature on garbage collection records a variety of object scanning mechanisms, but despite its performance-critical role, to our knowledge there has been no prior study quantitatively evaluating the various approaches. As we show here, a detailed understanding of these tradeoffs informs the design of the best performing object scanning mechanisms.

The mechanism for scanning an object typically involves parsing metadata that is explicitly or implicitly associated with the object. The means of parsing and the form of the metadata can vary widely from one implementation to another. We identify four major dimensions in the design space: i) compiled versus interpreted evaluation of metadata, ii) encoding and packing of metadata, iii) levels of indirection between each object and its metadata, and iv) variations in object layout.

To inform our study of design tradeoffs, we first perform a detailed analysis of heap composition and object structure as seen by the garbage collector. We conduct our study within Jikes RVM [Alpern et al., 2000], a high performance research JVM with a well tuned garbage collection infrastructure [Blackburn et al., 2004b]. First, to characterize the workload seen by any scanning mechanism, we execute eighteen benchmarks from the DaCapo [Blackburn et al., 2006] and SPEC [SPEC, 2001, 1999] suites, and at regular intervals examine the heap and establish the distribution of object layouts among traced objects. We were not surprised to find that a relatively small number of object layout patterns account for the vast majority of scanned objects. We include in this study the extent to which packing of reference fields within objects changes the distribution of layout patterns.

Guided by this information, we conduct a performance analysis of various object scanning implementation alternatives. We evaluate each alternative on four architectures against the DaCapo and SPEC suites. We observe substantial variation in performance among architectures but find that some mechanisms yield consistent, significant advantages, averaging 16% or more relative to a well tuned baseline. Specifically, we find that metadata encoding offers consistent modest advantages, object field reordering gives little measurable advantage (but improves the effectiveness of other

optimizations), and that specialized compiled scanning code for common cases significantly outperforms interpretation of metadata. The most effective scheme uses a small amount of metadata encoded cheaply into the object header to encode the most common object patterns.

We also implement and evaluate the bidirectional object layout used by SableVM [Gagnon and Hendren, 2001] and find that it performs well compared to orthodox object layout schemes. The Sable object model combined with specialization of object scanning code outperforms the alternatives in almost all benchmarks. There is however a small but consistent overhead in mutator time for this object model, giving it an advantage in overall time when the heap is small, and a slight disadvantage in large heaps.

This study is the first in-depth evaluation of object scanning techniques and the tradeoffs they are exposed to. As far as we know, our findings are the first to provide a quantitative foundation for the design and implementation of tracing, the performance-critical mechanism at the heart of all modern garbage collection implementations.

5.2 Related Work

Sansom [1991] appears to have been the first to propose compiling specialized code for scanning objects (see Section 5.4.3), although he did not perform a performance analysis of the benefits of this technique. Jones and Lins [1996], authors of the standard text on garbage collection, make reference to Sansom's work and subsequent work, but do not directly discuss the question of design options for scanning mechanisms. Grove and Cheng did a proof-of-concept implementation of scanning specialization for Jikes RVM and concluded that it was a profitable idea, but did not publish this work or incorporate it into the main code base [Grove and Cheng, 2005]. David Grove kindly provided us with their implementation, which we forward-ported and used as the basis for our implementation of specialization. This implementation has been the default scanning mechanism in Jikes RVM since 2007.

Gagnon carefully examined the question of object layout and garbage collection efficiency in his PhD thesis [Gagnon, 2002]. He proposed the bidirectional object layout, where reference and non-reference fields are laid out on opposite sides of the object header. SableVM implements this object layout [Gagnon and Hendren, 2001]. This design has two significant properties: a) it maintains separation of reference and non-reference fields in spite of accretion of fields due to inheritance, and b) object scanning logic is trivial since reference fields are always contiguous. Since SableVM did not have an optimizing compiler, it was hard for Gagnon to perform a detailed performance evaluation of this design. More recently Gu, Verbrugge, and Gagnon [2006] set out to compare the performance of this layout in Jikes RVM but concluded that it was difficult to accurately evaluate such design choices in the context of a complex, non-deterministic JVM. Dayong Gu generously made available to us his port to Jikes RVM of the SableVM bidirectional object model, which we forward-ported,

tuned, and used in our evaluation of the bidirectional object model reported here. We use replay compilation to remove the non-determinism of the adaptive optimization system and found significant, repeatable results across four architectures.

5.3 Analysis of Scanning Patterns

To ground our study of scanning mechanisms, we begin with a comprehensive analysis of the distribution of object layout patterns, as seen at garbage collection (GC) time for a large suite of benchmarks. Since scanning consists of identifying and then acting on the reference fields of objects transitively in the heap, understanding the distribution of the patterns in which reference fields occur is important to the design decisions.

We use the term *reference* to describe a language-level reference to an object. The live object graph is defined as the set of objects that are transitively *referenced* from some set of roots. By contrast, we use the term *pointer* as an implementation-level address ('`void *`') which may or may not point to an object. In practice a language may implement references as pointers but object liveness is nonetheless defined in terms of references, which in general is a subset of the pointers. We define the *reference pattern* of an object to be the number and location of reference fields within the object. All objects of a given class have the same reference pattern, and two classes may have the same pattern even though they differ in such aspects as size (in bytes), number of fields, or inheritance depth.

Because the policy for the layout of references within an object will affect the distribution of reference patterns, we consider two key object layout regimes; *declaration order*, and *references first*. These alternatives are straightforward design choices and were described in Etienne Gagnon's PhD work as 'naive' and 'traditional' layouts respectively [Gagnon, 2002]. In the first case object fields appear within the object in the order in which they are statically declared (with minor adjustments to ensure efficient packing in the face of alignment requirements). This is the approach used by Jikes RVM. In the second case, references are packed together before non-reference fields, at *each* level of the inheritance tree for each class. Note that for efficiency reasons, language implementations generally require that field offsets are *fixed* across an inheritance hierarchy, allowing the same code to access fields of a class and all of its subclasses. So in practice, the field layout for any subclass may only be additive with respect to its super class. Thus the 'references first' layout will typically result in alternating regions of references and non-references corresponding to levels of inheritance for the given type. Gagnon's *bidirectional* object layout [Gagnon, 2002] avoids this problem by growing the object layout in two directions, with references on one side and non-references on the other. Thus references will always be packed on one side of the object header regardless of inheritance.

A minor variant on the 'references first' scheme involves alternating the packing of reference fields first or last in an attempt to maximize the opportunity for contiguous groups of reference fields, and could in principle lead to further speedups. In prac-

tice we found that such schemes perform almost identically to the 'references first' scheme, and in the interests of space we omit any further discussion.

5.3.1 Analysis Methodology

In order to conduct our analysis of scanning patterns, we instrument Jikes RVM to identify and then record the reference pattern for each object that it scans at collection time. At the end of the execution of each benchmark, the collector prints a histogram indicating the frequency with which each reference pattern was seen by the scanning mechanism throughout the execution of the benchmark. We hold the collection workload constant by setting a fixed heap of $2\times$ the minimum heap size for each benchmark. This is a moderate heap size and is same size as we use in our performance study in Section 5.6. We chose $2\times$ as representative of a 'reasonable' heap, although our analysis is largely insensitive to heap size. If the heap were made significantly tighter, very short lived objects may be slightly more prominent, and of course if the heap were made considerably larger collections would happen less frequently or not at all, making our analysis more difficult.

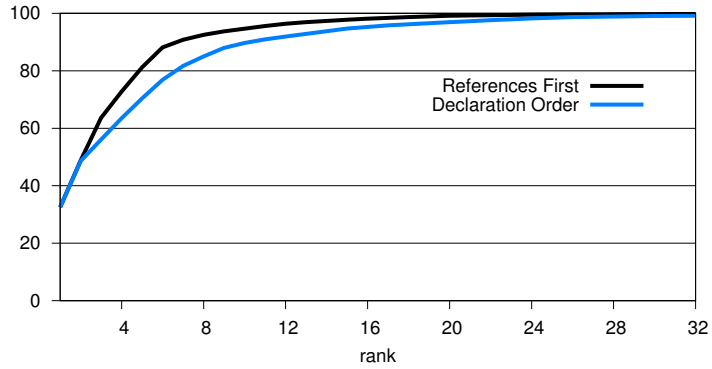
5.3.2 Encoding and Counting Patterns

We study the frequency distribution of reference layout patterns in objects. Since the number of different possible reference layouts is enormous, to make the study tractable, we consider a fixed set of $2^{16} + 4$ layouts. We exhaustively consider all 2^{16} layouts possible for non-array reference patterns of up to 16 words in length. We bound the set by grouping together all non-array reference patterns of 17-32 words in length, and all non-array reference patterns greater than 32 words in length. In practice, such patterns comprise just 0.58% and 0.10% of all objects respectively. Because all other patterns are counted precisely, our study is precise with respect to 99.32% of all objects for the benchmarks we study. The relative size of the pattern groups is as follows: a) objects with no references (29.60%), b) arrays of references (6.17%, 35.77% cumulatively), c) the 2^{16} reference layout patterns that can potentially arise in objects with up to 16 words in length (63.55%, 99.32% cumulatively), d) non-array objects with references that are 17–32 words in length (0.58%, 99.90% cumulatively), and e) non-array objects with references that are larger than 32 words in length (0.10%, 100% cumulatively).

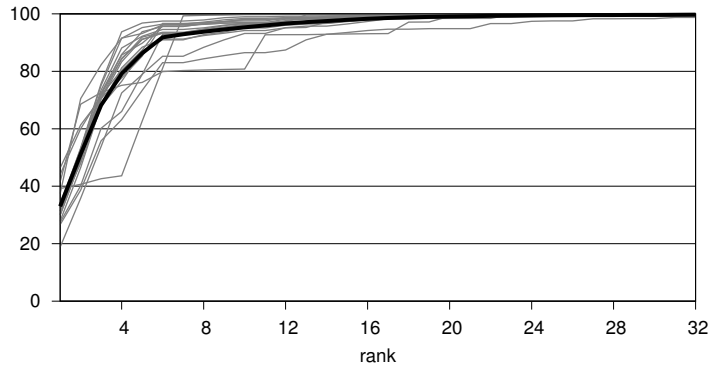
Our instrumentation works as follows. We modify Jikes RVM to encode each non-array object's reference pattern as a 32 bit vector in the per-class metadata. Each bit maps to a word in the object and identifies whether that word is a reference or not. For example, an object which contained (only) two references, in its first and third words, would be encoded as `0...0101` (0x5). An object with references (only) in the first, third and sixth fields would be encoded as `0...0100101` (0x25). We create a histogram with $2^{16} + 4$ bins (to account for each of our fixed set of reference layouts) and initialize the bins to zero at the start of execution. When each object is scanned during each garbage collection, we determine its pattern either as one of the four spe-

rank	reference pattern	mean	cumulative mean	jess	compress	raytrace	db	javac	mpegaudio	mtrt	jack	antlr	bloat	fop	hsqldb	jython	lusearch	luindex	pmd	xalan	jbb2000
1	no references	33.02	33.02	30.28	33.01	44.09	37.44	31.41	33.94	46.36	32.36	30.01	27.88	26.67	39.16	27.42	31.34	29.96	18.85	32.70	41.54
2	00 0000 0000 0001	18.19	51.21	21.65	20.32	15.93	33.10	19.05	19.14	15.01	20.15	18.85	18.62	11.99	1.53	12.79	20.72	19.24	16.97	15.27	27.05
3	00 0000 0000 0111	16.99	68.20	20.17	22.42	11.60	11.72	21.63	22.05	9.30	20.92	21.41	21.72	17.13	1.94	19.97	20.40	21.59	17.98	19.89	4.00
4	00 0000 0011 1111	10.95	79.15	13.54	17.96	8.78	9.31	12.13	16.36	7.00	14.69	13.04	13.00	7.41	1.03	5.85	13.41	13.74	18.67	8.63	2.53
5	00 0000 0000 0011	7.14	86.29	4.16	3.07	6.96	1.60	7.70	3.74	7.63	4.09	7.86	7.19	10.16	19.00	12.92	7.89	6.57	6.41	10.55	1.00
6	refarray	5.55	91.84	5.86	0.70	9.28	2.96	1.68	1.21	10.90	2.36	2.21	2.82	9.64	18.42	6.28	1.83	2.17	13.97	3.92	3.76
7	00 0000 0011 1101	1.03	92.88												18.22						0.41
8	00 0000 0111 0111	0.92	93.80	0.78	0.37	0.25	0.22	0.97	0.66	0.21	0.57	1.31	1.19	1.35	0.17	3.13	1.02	1.11	1.27	1.82	0.18
9	00 0000 0001 1011	0.80	94.59	0.89	0.82	0.40	0.44	0.79	0.80	0.32	0.90	1.21	0.85	1.13	0.12	2.58	0.63	0.88	0.59	0.86	0.15
10	00 0111 1001 1111	0.73	95.32	0.72	0.33	0.23	0.20	0.91	0.54	0.19	0.52	0.99	0.91	1.03	0.12	2.23	0.76	0.85	0.94	1.42	0.17
11	00 0000 0001 1101	0.66	95.98								0.02										11.93
12	00 0000 0000 1111	0.64	96.63	0.28	0.17	1.04	0.12	0.29	0.24	1.21	1.04	0.31	0.96	0.93	0.04	1.98	0.37	0.44	1.13	0.98	0.04
13	00 0000 0000 0010	0.47	97.10	0.16	0.01		2.51	0.01	0.01		0.52	0.01	0.14	3.50		0.57	0.06	0.37	0.26	0.40	
14	00 0001 0110 0001	0.40	97.50	0.01	0.01	0.50		0.11		0.96		0.54	1.81	1.98			0.01	0.85		0.28	0.12
15	00 0000 0001 1111	0.40	97.90	0.31	0.21	0.13	0.09	0.62	0.38	0.10	0.72	0.48	0.45	0.60	0.09	0.77	0.38	0.46	0.43	0.81	0.10
16	00 0000 0011 0111	0.36	98.26	0.30	0.19	0.11	0.08	0.60	0.35	0.09	0.32	0.52	0.46	0.66	0.09	0.87	0.36	0.42	0.40	0.64	0.10
17	00 0111 1100 0011	0.30	98.56	0.39	0.14	0.12	0.09	0.47	0.23	0.09	0.29	0.47	0.40	0.47	0.03	0.90	0.18	0.24	0.38	0.47	0.06
18	00 0000 1111 1101	0.22	98.78																		4.00
19	$>$ 31 bits	0.14	98.92	0.15	0.06	0.04	0.04	0.14	0.10	0.03	0.10	0.16	0.13	0.20	0.02	0.62	0.16	0.17	0.15	0.25	0.02
20	00 0000 0000 1101	0.13	99.06					0.39												0.01	2.00
21	00 0000 0000 0110	0.12	99.18	0.02	0.08	0.01	0.02	0.76	0.03	0.01	0.06	0.11	0.50	0.01		0.18	0.05	0.15	0.03	0.07	
22	$>$ 16 bits	0.11	99.29									0.03		1.77			0.02			0.14	
23	01 0000 0011 1111	0.07	99.36																1.31		
24	00 0001 1111 1111	0.06	99.42	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.03	0.01	0.79		0.07	0.02	0.05	0.01	0.01	
25	00 0000 1001 1111	0.06	99.47	0.02	0.01	0.01	0.01	0.01	0.05	0.01	0.02	0.09	0.09	0.10	0.02	0.05	0.09	0.09	0.15	0.19	
26	00 1110 0110 1111	0.05	99.53			0.43				0.52											
27	11 1111 0000 1111	0.04	99.57											0.78							
28	00 0000 0100 0011	0.03	99.60														0.12	0.40			
29	00 1110 1100 0011	0.03	99.63	0.02	0.01	0.02		0.10	0.05	0.02	0.03	0.04	0.02	0.01		0.04	0.03	0.03	0.02	0.07	0.01
30	00 0000 1101 1111	0.03	99.65													0.50					
31	01 1100 0001 1111	0.02	99.68											0.44							
32	00 0000 0000 0101	0.02	99.70	0.10				0.16			0.07										0.08

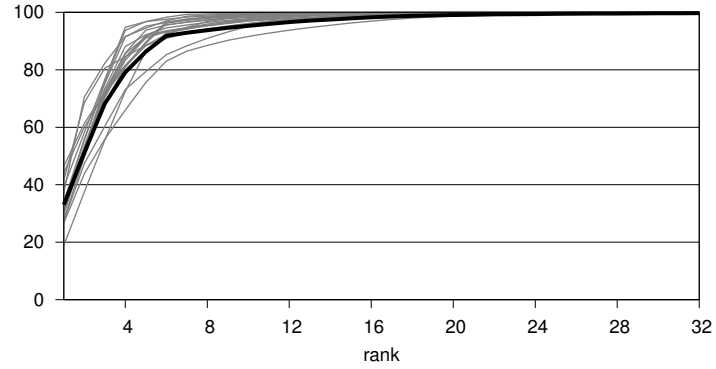
Table 5.1: Detailed *reference layout pattern* distributions for 'references first' object layout (all numbers expressed as percentages).



(a) 'References first' and 'declaration order'. Mean of all benchmarks.



(b) 'References first' per-benchmark distributions, global rank. Mean in Black.



(c) 'References first' per-benchmark distributions, per-benchmark rank. Mean in Black.

Figure 5.1: Cumulative frequency distribution curves for *reference layout patterns*. Each graph plots cumulative percentage of all objects (y-axis) covered by the N most common patterns (x-axis).

cial cases, or by using the low 16 bits of the object's encoding. We then increment the appropriate bin in the histogram. At the end of execution we print out the histogram.

We also inform our study of the Sable object layout by counting the number of reference fields in each object.

5.3.3 Benchmarks

We use the the DaCapo (version 2006-10-MR2) and SPECjvm98 benchmark suites, and jbb2000 in all of the measurements taken in this chapter. We did not use DaCapo eclipse because its use of classloaders is incompatible with Jikes RVM's replay mechanism. We did not use chart because of problems on 64-bit Ubuntu with the Java libraries that (only) chart depends on.

5.3.4 Reference Pattern Distributions

Table 5.1 and Figure 5.1 summarize the results of our study of scanning pattern distribution.

Figure 5.1a shows a cumulative frequency plot of scanning patterns. In this graph, the y-axis represents the percentage of all scanned objects covered by the number of patterns on the x-axis. The patterns are ordered from most to least coverage, so from left to right each additional pattern has a diminishing impact on the total coverage. The two curves in Figure 5.1a each plot the mean of all eighteen DaCapo and SPEC benchmarks. We show curves for both 'references first' and 'declaration order' object layouts.

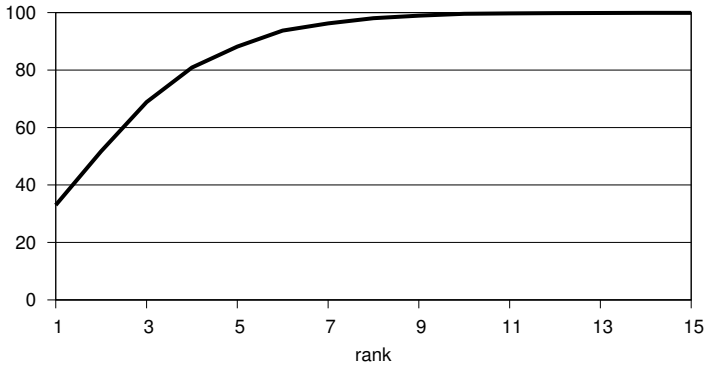
Table 5.1 shows the 32 'reference first' patterns which, when averaged over all eighteen benchmarks, have the highest coverage of object scans. The first column gives the rank importance, the second column shows a binary representation of the reference pattern (or identifies the special case), the third column states the percentage of scanned objects covered by the pattern (the mean of the per-benchmark percentages) and the fourth column gives the cumulative value of column three. Columns one and four correspond to the x and y axes of Figure 5.1a. The remaining columns give the percentage coverage for the pattern on each benchmark.

Figure 5.1a shows that by packing references together as much as possible, 'references first' requires significantly fewer patterns to cover a given number of objects. We find that of the large space of possible reference patterns, remarkably few are needed to cover the vast majority of scanned objects. Specifically, 6 (11) patterns cover 90% of scanned objects, 10 (16) patterns cover 95%, and 20 (30) patterns cover 99% for 'references first' and 'declaration order' object layouts respectively.

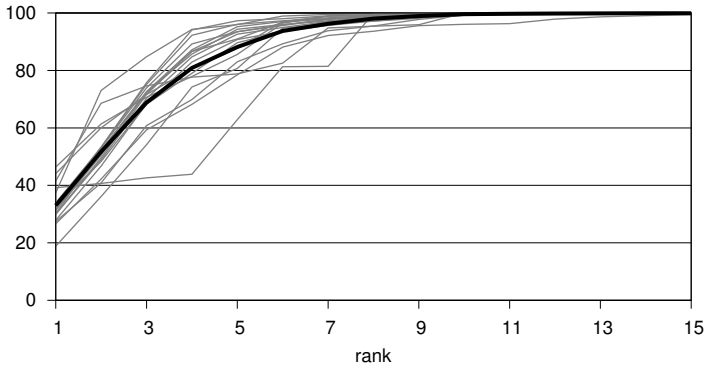
Figure 5.1b and columns five onward of Table 5.1 show the frequency distribution for each of the eighteen benchmarks using the 'references first' object layout. In Figure 5.1c, the cumulative total is separately calculated for each benchmark with respect to that benchmark's ordering of pattern importance. On the other hand, Table 5.1 and Figure 5.1b present the data using a single global ordering of patterns. Here we see that on a benchmark-by-benchmark basis, the situation is accentuated further, with

rank	pointer count	mean	cumulative mean	jess	compress	raytrace	db	javac	mpegaudio	mtrt	jack	antlr	bloat	fop	hsqldb	jython	lusearch	luindex	pmd	xalan	jbb2000
1	0	33.02	33.02	30.31	33.01	44.09	37.44	31.41	33.94	46.36	32.35	29.96	27.88	26.65	39.16	27.42	31.37	29.97	18.83	32.65	41.54
2	1	18.68	51.70	21.79	20.34	15.94	35.61	19.06	19.15	15.01	20.70	18.94	18.81	15.57	1.53	13.44	20.79	19.59	17.27	15.70	27.05
3	3	17.17	68.87	20.18	22.43	11.60	11.72	22.02	22.05	9.30	20.91	21.37	21.81	17.10	1.94	19.97	20.51	22.16	18.06	19.91	6.01
4	6	11.97	80.85	14.37	18.34	9.07	9.54	13.21	17.11	7.24	15.29	14.47	14.41	8.87	1.22	9.06	14.53	14.90	20.09	10.74	3.08
5	2	7.31	88.16	4.31	3.18	6.99	1.64	8.64	3.81	7.65	4.24	7.90	7.85	10.25	19.00	13.10	8.02	6.85	6.44	10.67	1.09
6	refarray	5.56	93.72	5.83	0.70	9.28	2.96	1.68	1.21	10.90	2.38	2.30	2.82	9.67	18.42	6.27	1.83	2.15	13.95	3.91	3.77
7	4	2.50	96.22	1.23	1.00	1.95	0.56	1.19	1.04	2.48	1.97	2.07	3.62	4.04	0.16	4.56	1.01	2.14	1.72	2.11	12.25
8	5	1.84	98.06	0.62	0.41	0.25	0.18	1.22	0.74	0.20	1.21	1.04	1.19	1.49	18.39	1.64	0.75	0.88	0.83	1.48	0.61
9	9	0.87	98.93	0.74	0.34	0.67	0.20	0.92	0.56	0.71	0.53	1.02	0.92	1.96	0.13	2.30	0.78	0.89	0.95	1.45	0.52
10	7	0.67	99.59	0.45	0.16	0.13	0.10	0.48	0.25	0.10	0.31	0.72	0.46	0.47	0.03	1.51	0.21	0.26	1.70	0.57	4.06
11	32	0.14	99.73	0.15	0.06	0.04	0.04	0.14	0.10	0.03	0.09	0.16	0.13	0.20	0.02	0.62	0.16	0.16	0.15	0.21	0.01
12	12	0.10	99.83											1.60		0.02				0.11	
13	10	0.04	99.87											0.78							
14	8	0.04	99.91						0.02				0.05	0.44		0.07				0.09	
15	15	0.02	99.93											0.33			0.02	0.04		0.06	

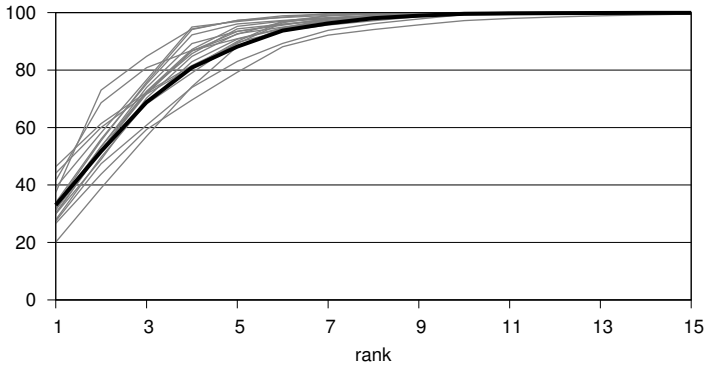
Table 5.2: Detailed reference *field count* distributions (all numbers expressed as percentages).



(a) Reference fields per object. Mean of all benchmarks.



(b) Reference fields per object, per-benchmark distributions, global rank. Mean in black.



(c) Reference fields per object, per-benchmark distributions, per-benchmark rank. Mean in black.

Figure 5.2: Cumulative frequency distribution curves for reference *field counts*. Each graph plots cumulative percentage of all objects (y-axis) covered by the N most common reference field counts (x-axis).

very few patterns required to cover most scanning cases. The left-most curve at the 80th percentile is for db, and the two left most at the 95th percentile are for compress and hsqldb. db requires just 4, 6, and 8 patterns to cover 90%, 95% and 99% of all scanned objects respectively. Only four benchmarks fall significantly below the mean, namely fop, jython, pmd and xalan. The most prominent outlier is fop, which requires 9, 14, and 21 patterns to cover 90%, 95% and 99% of scanned objects. Thus even at worst, very few patterns are required to cover the vast bulk of scanned objects.

The data in Figure 5.1 and Table 5.1 show that a few special cases and a small number of patterns cover the vast majority of objects scanned, and furthermore that these common patterns are very simple. This suggests that object scanning mechanisms which can optimize for these few scenarios may be very effective.

5.3.5 Reference Field Count Distributions

The bidirectional Sable object layout depends only on the *number* of reference fields in an object, because the pattern of references and non-references is fixed. Table 5.2 and Figure 5.2 show the frequency distribution of number of reference fields among our benchmarks. This data shows that the vast majority of objects in all benchmarks have a small number of reference fields. 93% or more of objects in all benchmarks have 6 or fewer reference fields or are reference arrays, and 99% of all objects have 12 or fewer reference fields. There are, however, some outliers: the xalan benchmark has some scalar objects with 46 reference fields. Figure 5.2b highlights the variation in frequency between benchmarks even in the most common patterns.

These figures demonstrate that optimizations that focus on objects with a small number of reference fields have significant potential, especially in the bi-directional object model where reference fields are contiguous.

5.4 Design Alternatives

We now discuss the primary design dimensions for object scanning. We begin our discussion with a description of the object scanning mechanism in Jikes RVM (as at version 3.1.0).

5.4.1 The Jikes RVM Scanning Mechanism

Figure 5.3 shows three user objects, A, B, and C, and Jikes RVM metadata associated with object A (metadata for B and C is omitted for clarity). If A and C were of the same type, they would both have pointers to the same metadata. Each object has a two-word header, one of which is a pointer to a *TIB* (type information block) for the object's class. The TIB incorporates a dispatch table, a pointer to a type (class) object and some other per-type metadata. The type object points to an array of *offsets*, indicating the location of reference fields within each instance of the type (class). Thus to scan A, the garbage collector must follow three indirections to reach the offsets array for A,

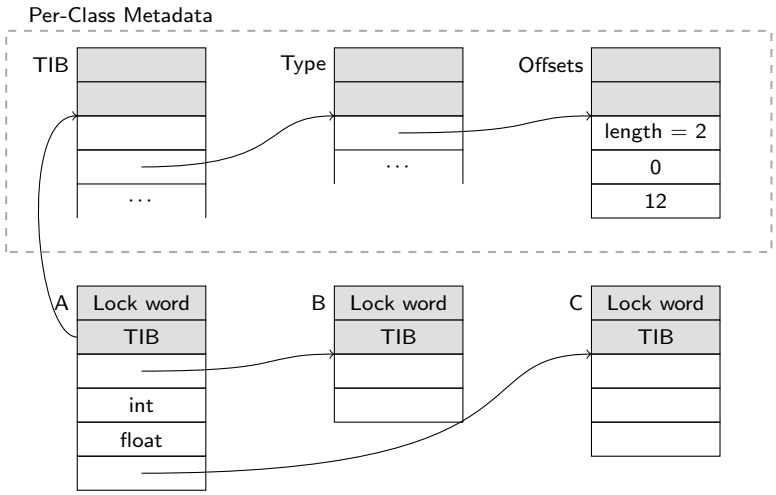


Figure 5.3: Objects and Per-Class Metadata Structure in Jikes RVM.

```
1  void scan(Object object) {
2    TIB tib = getTIB(object);
3    RVMType type = tib.getObjectType();
4    int[] offsets = type.getReferenceOffsets();
5    if (offsets != null) {
6      Address base = objectAsAddress(object);
7      for (int i=0; i < offsets.length; i++) {
8        processEdge(object,base.plus(offsets[i]));
9      }
10   } else { /* scan reference array */ }
11 }
```

Figure 5.4: The default scanning loop in Jikes RVM.

which identifies the location of each reference field. Figure 5.4 shows pseudocode for the default scanning code in Jikes RVM.

During tracing, Jikes RVM *'interprets'* each object's reference field layout by scanning the offset array. The offset array contains an entry for each reference field in a type, and encodes the offset (in bytes) to the given field from the object header. Jikes RVM makes no special effort to optimize object layouts for improved object scanning time.

5.4.2 Inlining Common Cases

One simple optimization is to hand-inline special case code for the most frequently executed patterns. This trades additional branches and code size for rare cases against faster execution of common cases. An example of this optimization when using offset arrays for scanning is given in Figure 5.5. Similar optimizations are possible alongside other design choices in scanning mechanism and object layout.

```
1  int[] offsets = type.getOffsets();
2  for (int i=0; i < offsets.length; i++) {
3    trace(obj, obj.plus(offsets[i]));
4  }
```

(a) Unoptimized scanning loop (using offset arrays).

```
1  static final int[] OFFSETS_ZERO = new int[0];
2  static final int[] OFFSETS_1 = new int[]{0};
3  static final int[] OFFSETS_7 = new int[]{0,4,8};
4  ...
5  int[] offsets = type.getOffsets();
6  // Optimized code for the frequent case
7  if (offsets == OFFSETS_ZERO) {
8    // Do nothing
9  } else if (offsets == OFFSETS_1) {
10   trace(obj,obj.plus(0));
11 } else if (offsets == OFFSETS_7) {
12   trace(obj,obj.plus(0));
13   trace(obj,obj.plus(4));
14   trace(obj,obj.plus(8));
15 } else {
16   for (int i=0; i < offsets.length; i++) {
17     trace(obj, obj.plus(offsets[i]));
18   } ...
```

(b) Optimized loop with hand-inlined code for patterns 0, 1 and 7.

Figure 5.5: Unoptimized and optimized versions of scanning code.

5.4.3 Compiled vs. Interpreted Evaluation

Sansom [1991] realized that a compiler could statically generate specialized code for scanning each type. This idea allows the garbage collector to use the standard dispatch method on each scanned object to execute code optimized for scanning that particular type, rather than interpreting metadata attached to the object. Advantages of this approach include a lower data cache footprint by removing the memory accesses to per-instance metadata, and avoiding branches associated with iteratively interpreting the metadata. On the other hand, this approach incurs a dynamic dispatch overhead and has a greater instruction cache footprint than interpreting. Variations on this approach may include specialization by object layout pattern rather than object type (removing redundancy and reducing instruction cache footprint), and limiting compilation to a modest number of common patterns (falling back to interpretation in all other cases).

5.4.4 Encoding and Packing of Metadata

The Jikes RVM mechanism uses a simple array of offsets to encode the location of reference fields in each type. Alternative encodings could be used, including a bitmap indicating which words are references. Hybrids are also possible, whereby a fixed size

bitmap is used in common cases, with a fallback to an offset array for types unable to fit in the bitmap. Packed representations may allow the metadata to be directly encoded in the object header in many cases, thereby avoiding any indirection to the metadata data structure for those objects whose metadata could fit in the header.

In any virtual machine implementation, space in the object header is generally at a premium. Adding a word to the object header for GC metadata is an option, but the performance cost due to increased heap pressure and decreased cache locality outweigh any possible gains. Jikes RVM makes eight bits available to MMTk, which uses four of those bits for the mark state (see Chapter 4 for details). Our implementation of the bi-directional object model uses an additional bit to identify the word as a non-pointer in order to allow the object header to be found when scanning the object (as per [Gagnon, 2002]), which leaves three bits for encoding metadata in our case.

There is an alternative approach (which we use in this study) that allows us to obtain these metadata bits 'for free'. We exploit the fact that the GC metadata is constant across all objects of a given class. By selectively aligning the TIB (vtable) of each class, we effectively encode metadata into the header field that stores the TIB pointer. We achieve this quite simply: when allocating a TIB and encoding n bits of metadata, we allocate a block of memory 2^n words larger than the TIB itself. Then we choose a start location within this chunk of memory that puts our metadata value into bits $w \ldots w + n - 1$, where w is the number of bits required to naturally align a pointer (i.e. $w = 2$ in a 32-bit machine). This scheme also has the advantage that it doesn't require an additional initializing store to the object header when an object is allocated. On a 32-bit machine we incur a space cost of 32 bytes per loaded class, which is insignificant.

5.4.5 Indirection to Metadata

The diagram of the Jikes RVM object metadata (Figure 5.3) indicates the potential to shorten the level of indirection from the object to its metadata. We look at the effects of removing one of these levels of indirection by allocating an additional field in the TIB for holding a pointer to the reference offsets array. We evaluate the cost of this in Section 5.6. Schemes where metadata is encoded into the object header also benefit from the absence of indirection, although we don't directly study the effects of this.

5.4.6 Object Layout Optimizations

In addition to increasing opportunities for commonality among distinct types (Section 5.3), object layout strategies can more directly impact object scanning performance. The bidirectional object layout proposed by Gagnon [Gagnon, 2002] and used in SableVM [Gagnon and Hendren, 2001] arranges every object so that reference fields are packed on one side of the object header, while non-reference fields are packed on the other. SableVM itself encodes the number of references into the object header, and in this study we look at the effects of the object layout separately from the effect of the header metadata optimization. As discussed in Section 5.3, an important property of

the bidirectional layout is that it maintains reference packing in the face of inheritance. Unidirectional field packing may offer some benefits, but sub-types must strictly append their declared fields, potentially interrupting any grouping of reference and non-reference fields inherited from the parent class. Unidirectional field packing may be profitable in hybrid schemes where common cases are handled differently. In these scenarios, a field packing algorithm such as 'references first' will increase the coverage of a given set of special cases (Section 5.3, Figure 5.1 and Table 5.1), thereby improving the efficacy of the special cases.

The potential drawback of the bi-directional object model is that there is no longer a fixed offset from the start of the memory region occupied by an object and its header/object pointer. Lazy sweeping in particular can be adversely affected by this, and we see this in the mutator time results in Figure 5.11c.

5.5 Methodology

The methodology used for our analysis work is described in Section 5.3.1. We extend that here to describe the methodology used to evaluate the performance of the various scanning mechanisms.

We implement each scanning mechanism in Jikes RVM. We isolate and measure the time spent in the garbage collector's *scanning phase* (transitive closure), thereby excluding the time taken to establish roots etc, which is unaffected by the scanning mechanism we evaluate here. On average, scanning takes up $\sim$80% of total garbage collection time, and takes time proportional to the size of the heap. In Section 5.6.6 we also evaluate the effect on total time. We measure each of the DaCapo and SPEC benchmarks, timing the second benchmark iteration in a 2$\times$ heap as described in Section 5.3.1, and using replay compilation to avoid non-determinism due to adaptive compilation. We use MMTk's inbuilt timers which separately report total time spent in each of the major GC phases, including scanning. We use Jikes RVM's 'FastAdaptive' builds, which remove assertion checks and fully optimize all code for the virtual machine (and hence the garbage collector), and incorporate execution profile data to further optimize the code in the virtual machine. Experiments were performed 6 times for each benchmark, with the average for each benchmark normalized to the performance of the base configuration on that benchmark. We report the geometric mean of this normalized value across all benchmarks. The graphs show error bars for a 90% confidence interval using Student's t-distribution.

We use as a baseline an optimized version of the original Jikes RVM implementation described in Section 5.4 (see Figure 5.3), and which we refer to in the remainder of the paper as **Off-3/Decl.** This configuration has three levels of indirection from the object to the offset array and uses the 'declaration order' object layout. Because this is the configuration to which all others are normalized, it does not appear explicitly in the graphs.

We use the MMTk mark-sweep collector for all results presented here. It has the fastest scanning performance of any full-heap collector in Jikes RVM, and therefore

Platform	Clock	DRAM	L1 D	L1 I	LLC
Atom D510	1.8GHz	4GB	32KB	32KB	1MB
Core i5 670	3.4GHz	4GB	64KB	64KB	4MB
Core 2 Duo E7600	3.1GHz	4GB	32KB	32KB	3MB
AMD Phenom II X6 1055T	2.8GHz	4GB	64KB	64KB	6MB

Table 5.3: Hardware platforms for scanning experiments.

Name	Primary Metadata	Indirections	Hand-inlining	Layout
Off-2/Decl	Offset Array	2	N	Decl
Off-3/Decl[a]	*Offset Array*	*3*	*N*	*Decl*
Off-3/Ref	Offset Array	3	N	Ref
Off-3+Inl/Ref	Offset Array	3	Y	Ref
Off-3/Sable	Offset Array	3	N	Sable
Count-3/Sable	32-bit count[b]	3	Y	Sable
Bmp-3/Decl	32-bit bitmap	3	Y	Decl
Bmp-3/Ref	32-bit bitmap	3	Y	Ref
Hdr[1R]/Ref	1-bit Header	1	Y	Ref
Hdr[1Z]/Ref	1-bit Header	1	Y	Ref
Hdr[2]/Ref	2-bit Header	1	Y	Ref
Hdr[3]/Decl	3-bit Header	1	Y	Decl
Hdr[3]/Ref	3-bit Header	1	Y	Ref
Hdr[3]+Spec/Ref	3-bit Header[c]	1	Y	Ref
Hdr[3]/Sable	3-bit Header	1	Y	Sable
Spec/Decl	Specialization	2	N	Decl
Spec/Ref	Specialization	2	N	Ref
Spec/Sable	Specialization	2	N	Sable

[a] Baseline configuration.
[b] Only possible with the Sable object layout.
[c] Falls back to specialization, and then to Offset-3.

Table 5.4: Configurations evaluated.

the results of speeding up the scanning process are most visible. The optimizations we present here apply to any precise garbage collector.

5.5.1 Hardware Platforms

We use four different hardware platforms in our analysis, described in detail in Table 5.3. The systems were running Linux 2.6.32 kernels with Ubuntu 10.04.1 LTS. All CPUs were operated in 64-bit mode, although Jikes RVM is a 32-bit application.

5.5.2 Configurations

For our performance results we evaluate 18 configurations combining features from the design space outlined in Section 5.4. The specific configurations evaluated are summarised in Table 5.4. The metadata representations we use are:

- An array of 32-bit offsets.

- A 32-bit count field (only applicable to the Sable object model).
- A 32-bit bitmap. Two special values indicate that the object is a reference array or cannot be described in 32 bits. This is necessarily held outside the object header.
- A 3-bit field in the object header. We use this to encode the six most common patterns in Table 5.1, interpreting results with a series of 'if' statements in the scanning code. The seventh value indicates a fallback to the more general case. When using the bi-directional object model we use this field to encode the five most frequent reference field counts. The coverage of this scheme for both object models is shown in Table 5.5.
- A 2-bit field in the object header, indicating whether the object is a reference array, has zero references, a single reference in position 1, or the fallback case. The coverage of this scheme for both object models is shown in Table 5.5.
- A 1-bit header field indicating whether the object is a reference array ('1R').
- A 1-bit header field indicating whether the object has no reference fields ('1Z').

	Layout		% Objects		
Bits	**Scheme**	**Patterns**	**Min.**	**Mean**	**Max.**
3	Ref-first	6 most common	79.9	91.8	97.5
	Sable	5 most common	81.0	93.0	98.9
2	Ref-first	0, 1, refarray	46.5	56.8	73.5
	Sable	0, 1, refarray	47.1	57.3	76.0
1	n/a	0	18.85	33.02	46.36
	n/a	refarray	0.70	5.55	18.42

Table 5.5: Header encoding: Percentage of objects covered by the schemes evaluated.

In the declaration order and references first object layouts, our specialization implementation compiles 66 specialized methods, covering all objects with reference fields in the first six positions, with an additional method for reference arrays and a fallback method for the fallback case. In the Sable object layout, we compile 18 specialized methods, covering objects with up to 16 reference fields plus reference arrays and the fallback case.

5.6 Results

We now evaluate the performance of the design space described in Section 5.4. Since our focus is on the scanning mechanism, and the designs we explore have little or no impact outside of the scanning loop (which typically dominates garbage collection performance), unless otherwise stated, we present the relative performance of the scanning loop alone. Since many of the design dimensions are independent, we evaluate many combinations of the design choices in order to help understand which combinations of choices are most profitable. In total we implemented and evaluated

around twenty five which combine multiple optimizations. We only report results for the most significant of these.

This section concludes with a summary of the best performing designs, and their impact on scanning time and total execution time.

5.6.1 Inlining Common Cases

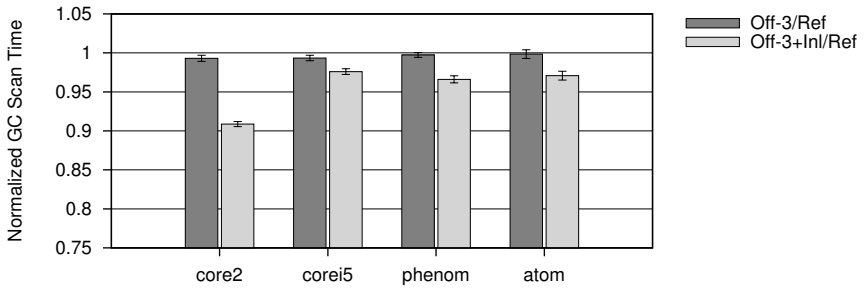


Figure 5.6: The effect of inlining common cases. Geometric mean of 18 benchmarks.

The speedup gained by hand-inlining the most frequently executed patterns (as described in Section 5.4.2) is illustrated in Figure 5.6, using Off-3/Ref and Off-3+Inl/Ref. The Off-3+Inl/Ref configuration uses the technique illustrated in Figure 5.5 to avoid interpreting the offset array for the most common object patterns. This shows that inlining common cases delivers a clear performance advantage. We use this technique in most of the configurations evaluated (the exceptions are identified in column four of Table 5.4).

5.6.2 Compiled vs. Interpreted Evaluation

In Figure 5.7 we compare specialized scanning (Section 5.4.3) across the three object layout schemes. Specialization performs well on average compared to the baseline

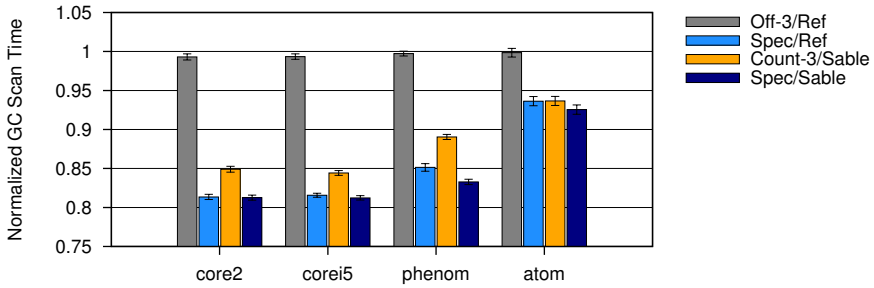


Figure 5.7: The effect of specialization. Geometric mean of 18 benchmarks.

Off-3/Decl configuration, but as we show in Section 5.6.6, not as well as three bits of header metadata. The reason is clear: for the 90% of objects that can be encoded by the 3-bit header field, scanning requires a load and then on average three conditional branches to the specialized code for scanning that object. Specialization requires two (dependent) loads and a jump, and on the Core i5 processor where an L1 cache hit costs four cycles, it is not difficult to see how this can be more expensive than the header metadata approach.

On the Atom processor, specialization offers less advantage, yielding a 7% speedup compared to a 15–18% speedup on the other processors. The out-of-order processors appear to be able to absorb more of the stall time caused by the indirect jump than the in-order Atom.

5.6.3 Encoding and Packing of Metadata

We now explore the header metadata design space described in Section 5.4.4.

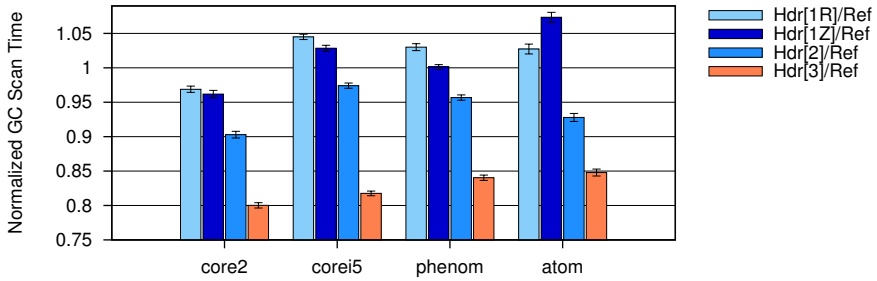


Figure 5.8: The effect of varying header encodings. Geometric mean of 18 benchmarks.

Figure 5.8 shows results for four configurations that use one, two, or three bits of header metadata. In each case we use the 'references first' object layout, and when optimizing special cases in the code we only optimize for the cases covered by the metadata. The 1-bit header fields reduce performance on all architectures except the Core 2. The 3-bit header field performs best, significantly outperforming the 2-bit header field, as predicted by the coverage figures given in Table 5.5.

5.6.4 Indirection to Metadata

In Figure 5.9 we explore the impact of indirection to metadata. The Off-2/Decl configuration differs only from the base Off-3/Decl configuration by one level of indirection. Since we can't practically build an Off-1/Decl configuration without adding a word to the object header, the graph also includes Hdr[3]/Decl which while not directly comparable, only uses one indirection to its metadata before applying its specific optimization.

The results show that shortening the path to the metadata achieves a modest 3–6% speedup, with the largest gain on the in-order Atom processor. Since Hdr[3]/Decl is the

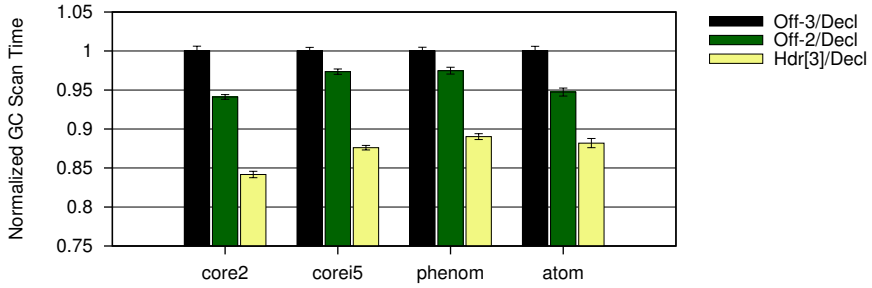


Figure 5.9: The effect of different levels of indirection. Geometric mean of 18 benchmarks.

result of removing one further level of indirection before applying the optimization evaluated in Section 5.6.1, we can see that the majority of Hdr[3]/Decl's speedup over Off-3/Decl is due to the elimination of indirection versus hand-inlining.

5.6.5 Object Layout Optimizations

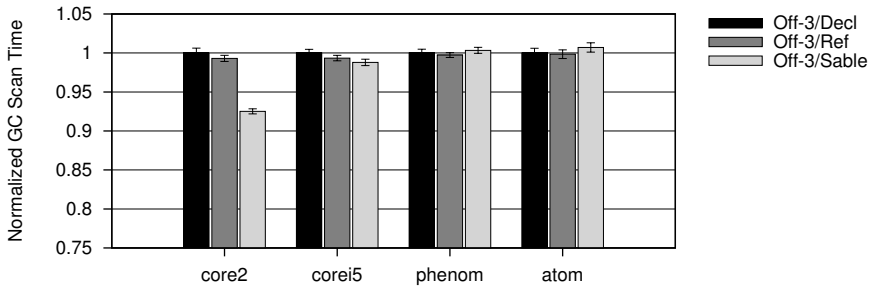
In this section we investigate the effect of changing the object layouts, both in the context of the default scanning mechanism, as well as interactions with design choices across the other dimensions. Figure 5.10 compares ten configurations, illustrating the effect of object layout on four different schemes.

Figure 5.10(a) shows that except on the Core 2 (an anomaly that remains unexplained), the choice of 'references first' or Sable object layout has very little impact on performance in the absence of any other optimizations. The slight improvement in performance on the out-of-order processors might be explained by small locality improvements.

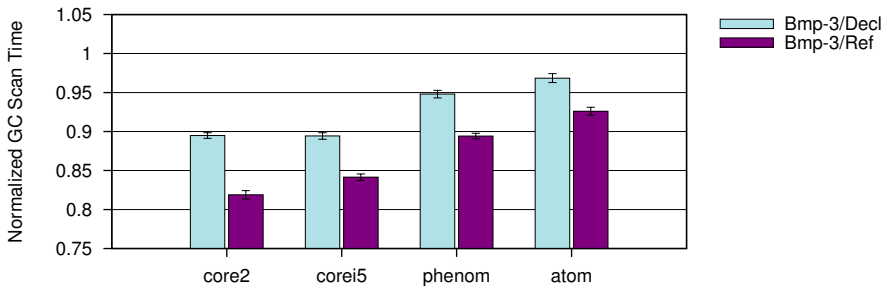
The graphs in Figures 5.10(b), (c) and (d) show that where another optimization is used, object layout has a significant impact on the effectiveness of the optimization. In all these cases the 'references first' layout improves significantly over the 'declaration order' object layout, while the Sable layout provides a small improvement over 'references first'.

5.6.6 Conclusion

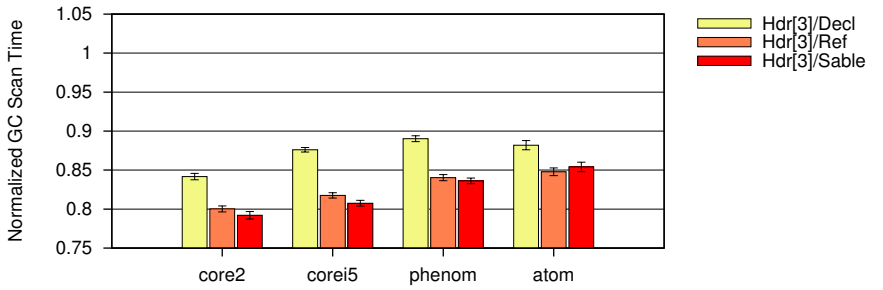
Figures 5.11a–(d) show the scan time and total time performance of six of the best performing designs. The performance of some design choices is highly affected by architecture. A bitmap performs poorly on the in-order Atom processor, as does specialized scanning. The combination Hdr[3]+Spec/Ref performs best on all architectures (within the limits of experimental error). The 3-bit field in the object header is a universally beneficial optimization when coupled with an object model that enhances its effectiveness. However, the per-benchmark results for these six designs shown in Figures 5.12–5.13 show that for benchmarks like hsqldb—where the object demographics



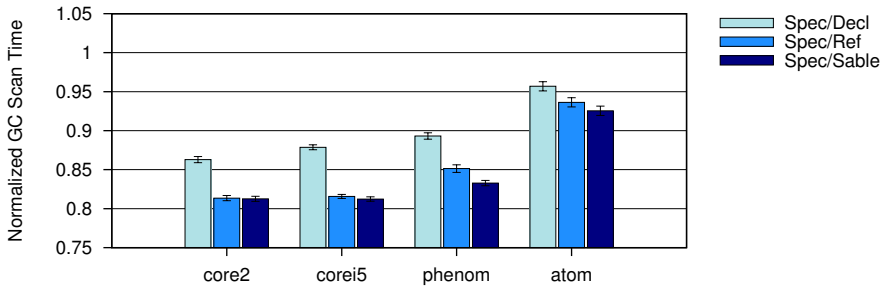
(a) Offset Array.



(b) Bitmap.

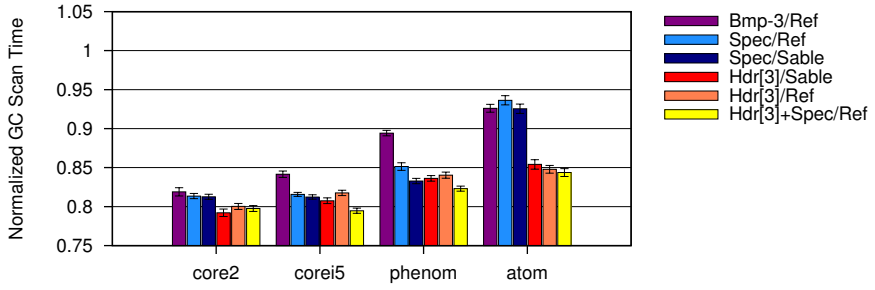


(c) Header Metadata.

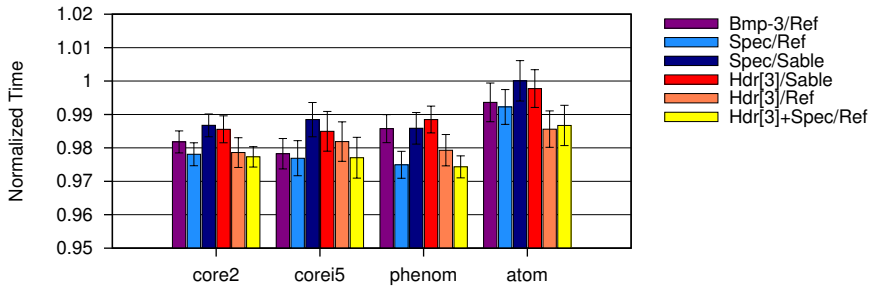


(d) Specialization.

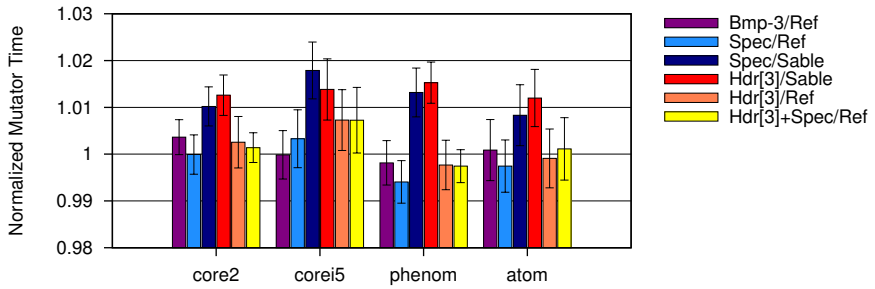
Figure 5.10: The effect of various object layout optimizations. Geometric mean of 18 benchmarks.



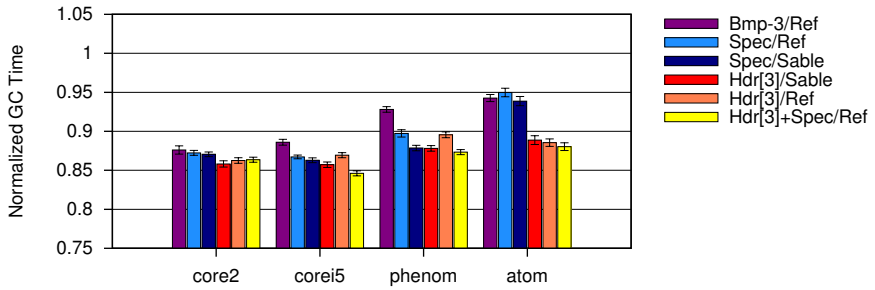
(a) Scan time, geometric mean.



(b) Total time, geometric mean.



(c) Mutator time, geometric mean.



(d) GC time, geometric mean.

Figure 5.11: Summary, showing four metrics for six well-performing designs.

are not a good match for the assumptions underlying the Hdr[3]/Decl configuration—Spec/Ref has a measurable advantage due its more comprehensive coverage of object patterns. Hdr[3]+Spec/Ref also performs well in this case because its less expensive fallback provides a 'soft landing' for these edge cases.

Figure 5.11b shows the effect of optimizations on total time. While the magnitude of the improvement is modest due to our choice of heap size, the Sable object model is less effective than the others due to a slight increase in mutator time. Nonetheless, these results show clearly that the choice of scanning design has a measurable effect on total execution time.

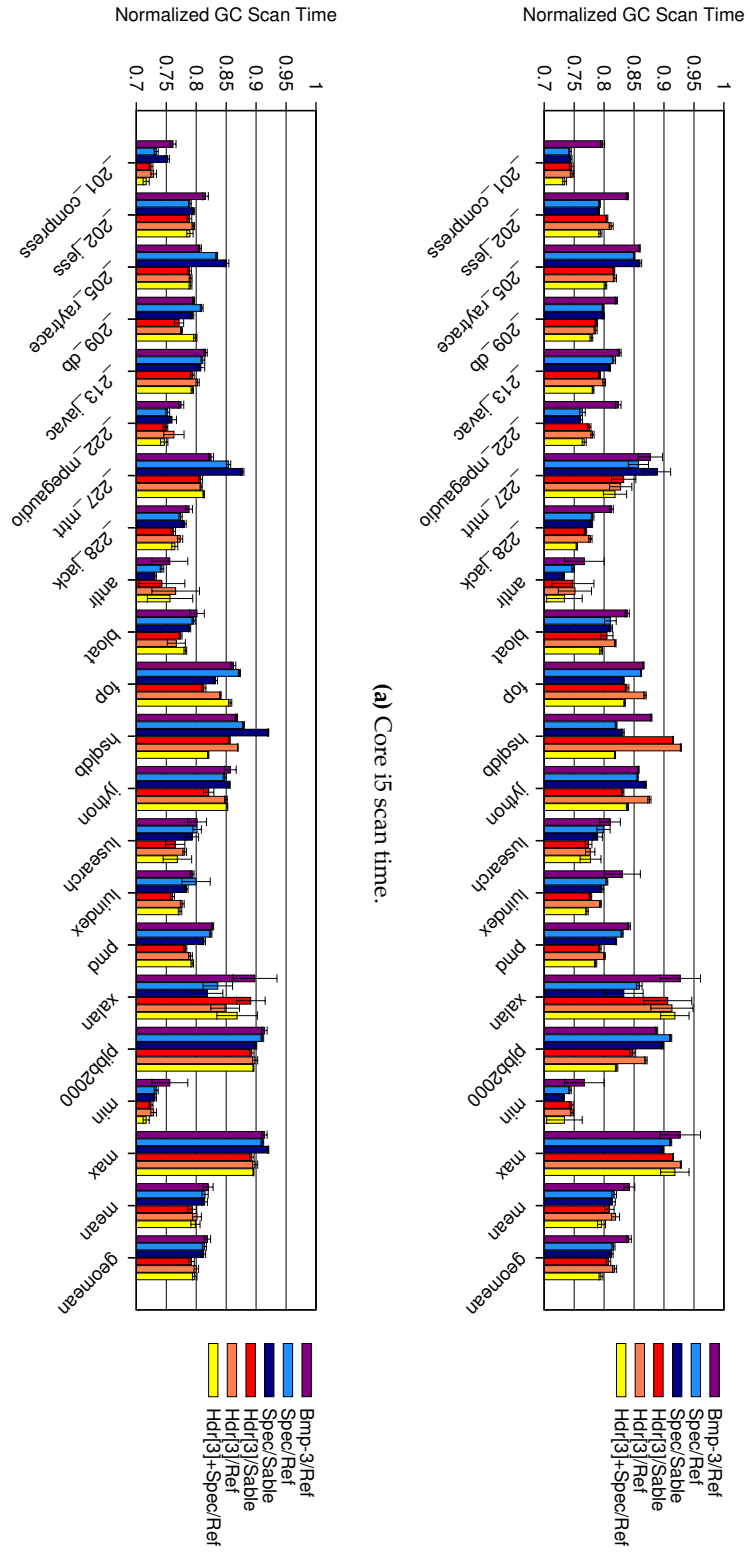
The important features of a high performance scanning mechanism (at least on the architectures we have benchmarked) are: a) the elimination of memory loads, both through indirection to metadata and in the metadata itself (see the performance of Bmp-3/Decl vs. Off-3/Decl); b) good choice of object layout, to facilitate the performance of the optimizations used; and c) good coverage of reference patterns.

The Hdr[3]+Spec/Ref design combines the best effects of all the optimizations discussed here. The 3-bit header field eliminates loads for the majority of objects, while 64 specialized patterns as a fallback provide good performance for benchmarks like hsqldb which are a poor match for the 3-bit field. This configuration achieves speedups in scan time of over 25% on several benchmarks, at no cost to mutator performance.

5.7 Summary

Object scanning is the mechanism at the heart of tracing garbage collectors. A number of object scanning mechanisms have been described in the literature, but—despite their performance-critical role—we are unaware of any prior work that provides a comprehensive study of their performance. In this chapter we outline the design space for object scanning mechanisms, and then use a comprehensive analysis of heap composition and object structure as seen by the garbage collector to inform key design decisions. We implement a large number of object scanning mechanisms, and measure their performance across a wide range of benchmarks. We include an implementation and evaluation of the bidirectional object layout used by SableVM [Gagnon and Hendren, 2001], and find that it performs well at collection time (although not significantly better than the more orthodox 'references first' optimized layout) but comes at a small but measurable cost to mutator performance. Our study shows that careful choice of object scanning mechanism alone can improve average scanning performance against a well tuned baseline by 21%, leading to a 16% reduction in GC time and an improvement of 2.5% in total application time in a moderate sized heap.

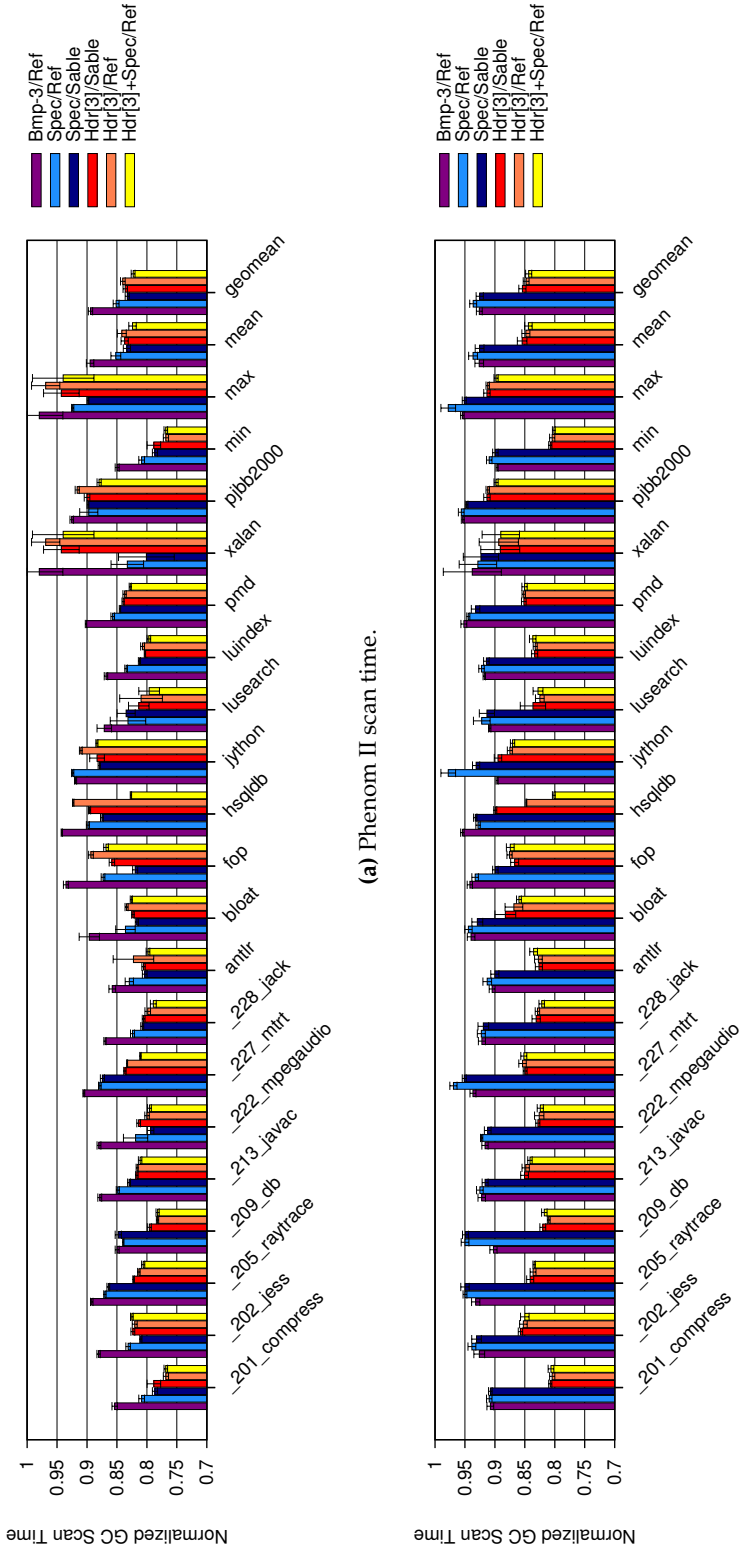
This chapter concludes our study of the performance of the garbage collection process. In Chapter 3 we analysed the performance of the garbage collector, and identified two major costs, marking and scanning. In Chapter 4 we looked at the first of these, and in this chapter we looked at the second. The next chapter concludes the thesis with a summary of the main results, and some directions for future work.



(a) Core i5 scan time.

(b) Core 2 scan time.

Figure 5.12: Per-benchmark scan times for six well-performing designs. (Core i5 and Core 2)



(a) Phenom II scan time.

(b) Atom D510 scan time.

Figure 5.13: Per-benchmark scan times for six well-performing designs. (Phenom and Atom processors)

Chapter 6

Conclusion

As managed runtimes for languages like Java and C# become more widely used, and as the memory footprints of the applications they run become larger, the efficiency of their garbage collectors becomes more important.

This thesis show that attention to the implementation details of a garbage collector can yield significant speedups. This in turn results in improved throughput, lower response times and increased acceptance of the languages themselves, with the associated software engineering benefits this brings.

The performance analysis technique of *replay tracing* allows garbage collector implementers to explore performance tradeoffs in isolation without having to fully implement and debug a complex change. We have used this technique to break down the costs of the garbage collection loop in a way that leads directly to performance optimizations. We have also used it to look at the performance impact of the garbage collector's traversal order of the heap, without interference from extraneous factors such as queue implementation details.

The *software prefetch* technique in Chapter 4 builds on insights gained from the replay tracing study to show that significant speedups are possible. A new implementation allows mark state to be kept in the object header, while minimizing floating garbage and using lazy sweeping. When we combine this with buffered prefetch and edge enqueuing, we can achieve significant, consistent speedups.

The survey of object scanning techniques in Chapter 5 demonstrates that there are significant speedups to be obtained from careful choice of object metadata representation. While considerable infrastructure is required to implement the fastest technique we demonstrate, there are significant speedups to be had from some simple changes.

6.1 Future Work

As all the results presented here show, processor architecture has a significant effect on garbage collector performance. Every time a new processor is released, new results may emerge from our analyses.

6.1.1 Prefetch

While we have effectively applied software prefetch to non-copying collection, the techniques we used to achieve this are not applicable to copying collectors. Indeed, it may be that copying collection does not benefit from prefetch, but it would be good to establish this definitively.

There are other garbage collectors and GC operations that might benefit from prefetch. Remembered sets in a generational collector are processed in a sequential manner, and it may be simple to speed this up using prefetch operations. Various operations in a reference counting collector (e.g. applying increments and decrements in a deferred reference counting collector) could benefit from prefetch.

6.1.2 Scanning

With respect to object scanning techniques, we have taken a purely static approach to all optimizations we have considered. Our optimizations perform less well on workloads that diverge significantly from the mean to which they are tuned. Ideally our implementation would dynamically adapt to the object demographics of the running application.

Our foray into specialized methods only looked at object scanning. For copying collectors (including nursery collections in generational collectors), specialization of copy methods is another plausible option.

Encoding more bits into the object header may have the potential for further speedups. 8 bits would encode a bitmap of over 99% of observed objects, but at a cost of 1–2KB per class depending on architecture. A specialized allocator that could reuse the unused spaces left after aligning the TIBs could make this both fast and space-efficient.

Bibliography

ALPERN, B., ATTANASIO, C. R., COCCHI, A., LIEBER, D., SMITH, S., NGO, T., BARTON, J. J., HUMMEL, S. F., SHEPERD, J. C., AND MERGEN, M., 1999. Implementing Jalapeño in Java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 314–324. ACM, New York, NY, USA. ISBN 1-58113-238-7 (cited on page 16)

ALPERN, B., ATTANASIO, D., BARTON, J. J., BURKE, M. G., CHENG, P., CHOI, J.-D., COCCHI, A., FINK, S. J., GROVE, D., HIND, M., HUMMEL, S. F., LIEBER, D., LITVINOV, V., MERGEN, M., NGO, T., RUSSELL, J. R., SARKAR, V., SERRANO, M. J., SHEPHERD, J., SMITH, S., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J., 2000. The Jalapeño virtual machine. *IBM System Journal*, 39(1) (cited on pages 9, 16, and 54)

APPEL, A. W., 1989. Simple Generational Garbage Collection and Fast Allocation. *Software–Practice and Experience*, 19(2):171–183 (cited on pages 15 and 18)

ARNOLD, M., FINK, S. J., GROVE, D., HIND, M., AND SWEENEY, P., 2000. Adaptive Optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 47–65. Minneapolis, MN (cited on page 16)

BACON, D. F., ATTANASIO, C. R., LEE, H. B., RAJAN, V. T., AND SMITH, S., 2001. Java without the coffee breaks: a nonintrusive multiprocessor garbage collector. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 92–103. ACM, New York, NY, USA. ISBN 1-58113-414-2 (cited on page 12)

BACON, D. F. AND RAJAN, V. T., 2001. Concurrent Cycle Collection in Reference Counted Systems. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 207–235. Springer-Verlag, London, UK. ISBN 3-540-42206-4 (cited on page 12)

BAKER, H. G., 1992. The treadmill: real-time garbage collection without motion sickness. *SIGPLAN Not.*, 27(3):66–70. ISSN 0362-1340 (cited on page 18)

BARTLETT, J. F., 1988. Compacting Garbage Collection with Ambiguous Roots. Technical Report WRL-88-2, DEC WRL (cited on page 16)

BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R., 2000. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings*

of the ninth international conference on Architectural support for programming languages and operating systems, ASPLOS-IX, pages 117–128. ACM, New York, NY, USA. ISBN 1-58113-317-0
URL http://doi.acm.org/10.1145/378993.379232 (cited on page 9)

BLACKBURN, S. M., CHENG, P., AND MCKINLEY, K. S., 2004a. Myths and realities: the performance impact of garbage collection. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 25–36. ACM, New York, NY, USA. ISBN 1-58113-873-3 (cited on pages 1, 2, 9, and 18)

BLACKBURN, S. M., CHENG, P., AND MCKINLEY, K. S., 2004b. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 137–146. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-2163-0 (cited on page 54)

BLACKBURN, S. M., GARNER, R., HOFFMANN, C., KHANG, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., MOSS, B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B., 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190. ACM, New York, NY, USA. ISBN 1-59593-348-4 (cited on pages 3, 21, 22, and 54)

BLACKBURN, S. M. AND MCKINLEY, K. S., 2003. Ulterior reference counting: fast garbage collection without a long wait. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 344–358. ACM, New York, NY, USA. ISBN 1-58113-712-5 (cited on page 12)

BLACKBURN, S. M. AND MCKINLEY, K. S., 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 22–32. ACM, New York, NY, USA. ISBN 978-1-59593-860-2 (cited on page 14)

BLACKBURN, S. M., MCKINLEY, K. S., GARNER, R., HOFFMANN, C., KHANG, A. M., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., MOSS, B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B., 2008. Wake up and smell the coffee. In *Communications of the ACM*, pages 83–89. ACM, New York, NY, USA (cited on pages 3 and 21)

BOEHM, H.-J., 1993. Space efficient conservative garbage collection. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design*

and implementation, pages 197–206. ACM, New York, NY, USA. ISBN 0-89791-598-4 (cited on page 54)

BOEHM, H.-J., 2000. Reducing Garbage Collector Cache Misses. In *The 2000 International Symposium on Memory Management*, pages 59–64
URL http://citeseer.ist.psu.edu/boehm00reducing.html (cited on pages 11, 23, 31, 42, 44, 45, 47, and 50)

BOEHM, H.-J., 2004. The space cost of lazy reference counting. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 210–219. ACM, New York, NY, USA. ISBN 1-58113-729-X
URL http://doi.acm.org/10.1145/964001.964019 (cited on page 12)

BOEHM, H.-J., 2012. A garbage collector for C and C++. Accessed May 2012
URL http://www.hpl.hp.com/personal/Hans_Boehm/gc/ (cited on page 19)

BOEHM, H.-J. AND WEISER, M., 1988. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820. ISSN 0038-0644 (cited on pages 15, 19, and 54)

CHENEY, C. J., 1970. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678. ISSN 0001-0782 (cited on page 13)

CHER, C.-Y., HOSKING, A. L., AND VIJAYKUMAR, T. N., 2004. Software prefetching for mark-sweep garbage collection: hardware analysis and software redesign. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 199–210. ACM Press, New York, NY, USA. ISBN 1-58113-804-0 (cited on pages xi, 23, 34, 36, 42, 43, 47, 48, 50, 51, and 52)

CHRISTOPHER, T. W., 1984. Reference Count Garbage Collection. *SPE*, 14(6):503–507 (cited on page 12)

CLINGER, W. D. AND HANSEN, L. T., 1997. Generational garbage collection and the radioactive decay model. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, pages 97–108. ACM, New York, NY, USA. ISBN 0-89791-907-6
URL http://doi.acm.org/10.1145/258915.258925 (cited on page 14)

COLLINS, G. E., 1960. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657. ISSN 0001-0782 (cited on page 11)

DEUTSCH, L. P. AND BOBROW, D. G., 1976. An efficient, incremental, automatic garbage collector. *Commun. ACM*, 19(9):522–526. ISSN 0001-0782 (cited on page 12)

DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M., 1976. On-The-Fly Garbage Collection: An Exercise in Cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York (cited on pages 7 and 10)

FENICHEL, R. R. AND YOCHELSON, J. C., 1969. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12:611–612. ISSN 0001-0782 (cited on page 12)

FRAMPTON, D., 2010. *Garbage Collection and the case for high-level low-level programming*. Ph.D. thesis, Department of Computer Science, Australian National University, Canberra, ACT (cited on page 12)

FRAMPTON, D., BLACKBURN, S. M., CHENG, P., GARNER, R. J., GROVE, D., MOSS, J. E. B., AND SALISHEV, S. I., 2009. Demystifying magic: high-level low-level programming. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 81–90. ACM, New York, NY, USA. ISBN 978-1-60558-375-4 (cited on page 17)

FRAMPTON, D., BLACKBURN, S. M., QUINANE, L. N., AND ZIGMAN, J., 2008. Cycle Tracing: Efficient Concurrent Cyclic Garbage. Will become a TR (cited on page 12)

GAGNON, E., 2002. *A Portable Research Framework for the Execution of Java Bytecode*. Ph.D. thesis, McGill University, Montreal (cited on pages 55, 56, and 66)

GAGNON, E. AND HENDREN, L., 2001. SableVM: A Research Framework for the Efficient Execution of Java Bytecode. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, Monterey, CA, USA*, pages 27–40. USENIX (cited on pages 55, 66, and 75)

GARNER, R., BLACKBURN, S. M., AND FRAMPTON, D., 2007. Effective prefetch for mark-sweep garbage collection. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 43–54. ACM, New York, NY, USA. ISBN 978-1-59593-893-0 (cited on pages 19, 23, and 41)

GARNER, R., BLACKBURN, S. M., AND FRAMPTON, D., 2011. A Comprehensive Evaluation of Object Scanning Techniques. In *ACM International Symposium on Memory Management* (cited on page 53)

GARTHWAITE, A. AND WHITE, D., 1998. The GC Interface in the EVM. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA (cited on page 9)

GCJ, 2012. *The GNU Compiler for the Java Programming Language*. The Free Software Foundation. Accessed May 2012
URL http://gcc.gnu.org/java/ (cited on page 19)

GEORGES, A., BUYTAERT, D., AND EECKHOUT, L., 2007. Statistically rigorous Java performance evaluation. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '07, pages 57–76. ACM, New York, NY, USA. ISBN 978-1-59593-786-5
URL http://doi.acm.org/10.1145/1297027.1297033 (cited on page 22)

GRONINGEN, J. V., 2004. Faster garbage collection using prefetching. In *Proceedings of Sixteenth International WOrkshop on Implementation and Application of Functional Languages (IFL04*, pages 142–152 (cited on page 43)

GROVE, D. AND CHENG, P., 2005. Private communication (cited on page 55)

GU, D., VERBRUGGE, C., AND GAGNON, E. M., 2006. Relative factors in performance analysis of Java virtual machines. In *VEE '06: Proceedings of the Second International Conference on Virtual Execution Environments*, pages 111–121. ACM Press, New York, NY, USA. ISBN 1-59593-332-6 (cited on pages 37 and 55)

HALLBERG, J., 2003. *Optimizing Memory Performance with a JVM: Prefetching in a Mark-and-Sweep Garbage Collector*. Master's thesis, Royal Institute of Technology, Kista, Sweden
URL http://web.it.kth.se/~matsbror/exjobb/msc_theses/josefinh_thesis_final.pdf (cited on pages 14, 34, and 43)

HANSEN, W. J., 1969. Compact list representation: definition, garbage collection, and system implementation. *Communications of the ACM*, 12(9):499–507 (cited on page 12)

HENNESSY, J. L. AND PATTERSON, D. A., 2006. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 0123704901 (cited on pages 20 and 53)

HERTZ, M. AND BERGER, E. D., 2005. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 313–326. ACM, New York, NY, USA. ISBN 1-59593-031-0
URL http://doi.acm.org/10.1145/1094811.1094836 (cited on page 6)

HICKS, M. W., MOORE, J. T., AND NETTLES, S., 1997. The Measured Cost of Copying Garbage Collection Mechanisms. In *ACM International Conference on Functional Programming*, pages 292–305 (cited on page 24)

HUANG, X., BLACKBURN, S. M., MCKINLEY, K. S., MOSS, J. E. B., WANG, Z., AND CHENG, P., 2004. The Garbage Collection Advantage: Improving Mutator Locality. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. Vancouver, BC (cited on pages 17 and 46)

HUGHES, R. J. M., 1982. A semi-incremental garbage collection algorithm. *Software - Practice and Experience*, 12:1081–1082 (cited on pages 11 and 42)

JONES, R. E., HOSKING, A. L., AND MOSS, J. E. B., 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman and Hall/CRC. ISBN 978-1420082791 (cited on page 5)

JONES, R. E. AND LINS, R. D., 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley (cited on pages 5, 44, 47, 54, and 55)

KNOWLTON, K. C., 1965. A fast storage allocator. *Commun. ACM*, 8:623–624. ISSN 0001-0782
URL http://doi.acm.org/10.1145/365628.365655 (cited on page 9)

LEA, D., 1998. A memory allocator
URL http://g.oswego.edu/dl/html/malloc.html (cited on page 9)

LEVANONI, Y. AND PETRANK, E., 2001. An on-the-fly reference counting garbage collector for Java. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 367–380. ACM, New York, NY, USA. ISBN 1-58113-335-9 (cited on page 12)

LIEBERMAN, H. AND HEWITT, C., 1983. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429. ISSN 0001-0782 (cited on page 14)

LINS, R. D., 1992. Cyclic reference counting with lazy mark-scan. *Inf. Process. Lett.*, 44(4):215–220. ISSN 0020-0190 (cited on page 12)

MARTÍNEZ, A. D., WACHENCHAUZER, R., AND LINS, R. D., 1990. Cyclic reference counting with local mark-scan. *Inf. Process. Lett.*, 34(1):31–35. ISSN 0020-0190 (cited on page 12)

MCCARTHY, J., 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM*, 3(4):184–195. ISSN 0001-0782 (cited on pages 1, 5, 7, 10, and 42)

OSSIA, Y., BEN-YITZHAK, O., GOFT, I., KOLODNER, E. K., LEIKEHMAN, V., AND OWSHANKO, A., 2002. A parallel, incremental and concurrent GC for servers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 129–140. ACM, New York, NY, USA. ISBN 1-58113-463-0 (cited on page 34)

QUINANE, L., 2003. *An Examination of Deferred Reference Counting and Cycle Detection*. Honours thesis, Australian National University
URL http://eprints.anu.edu.au/archive/00002710/ (cited on page 12)

RITZAU, T., 2003. *Memory Efficient Hard Real-Time Garbage Collection*. Ph.D. thesis, Linkpings Universitet (cited on page 12)

ROBSON, J. M., 1975. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20(3):242–244
URL http://comjnl.oxfordjournals.org/content/20/3/242.abstract (cited on page 8)

ROSE, J., 2008. Three software proverbs
URL http://blogs.oracle.com/jrose/entry/three_software_proverbs (cited on page 5)

ROVNER, P., 1985. On adding garbage collection and runtime types to a strongly typed, statically-checked, concurrent language. Technical Report Technical report CSL-84-7, Xerox PARC, Palo Alto, CA (cited on page 6)

SANSOM, P., 1991. Dual-Mode Garbage Collection. In H. Glaser and P. H. Hartel, editors, *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, pages 283–310. Department of Electronics and Computer Science, University of Southampton, Southampton, UK
URL http://citeseer.ist.psu.edu/sansom91dualmode.html (cited on pages 14, 55, and 65)

SPEC, 1999. *SPECjvm98 Documentation*. Standard Performance Evaluation Corporation, release 1.03 edition (cited on pages 20 and 54)

SPEC, 2001. *SPECjbb2000 (Java Business Benchmark) Documentation*. Standard Performance Evaluation Corporation, release 1.01 edition (cited on pages 20 and 54)

SPEC, 2006. *SPECjbb2005 (Java Business Benchmark) Documentation*. Standard Performance Evaluation Corporation, release 1.07 edition (cited on page 20)

SPEC, 2008. *SPECjvm2008 Documentation*. Standard Performance Evaluation Corporation, release 1.01 edition (cited on page 20)

STYGER, P., 1967. LISP 2 garbage collector specifications. Technical Report Technical Report TM-3417/500/00 1, System Development Cooperation (cited on page 13)

TRIDGELL, A., 2004. Using talloc in Samba4. Technical report, Samba Team
URL http://samba.org/ftp/unpacked/talloc/talloc_guide.txt (cited on page 6)

UNGAR, D., 1984. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 157–167. ACM, New York, NY, USA. ISBN 0-89791-131-8 (cited on page 14)

WEIZENBAUM, J., 1963. Symmetric list processor. *Commun. ACM*, 6:524–536. ISSN 0001-0782
URL http://doi.acm.org/10.1145/367593.367617 (cited on page 12)

WILSON, P. R., 1992. Uniprocessor Garbage Collection Techniques. In *IWMM '92: Proceedings of the International Workshop on Memory Management*, pages 1–42. Springer-Verlag, London, UK. ISBN 3-540-55940-X (cited on page 5)

WILSON, P. R., JOHNSTONE, M. S., NEELY, M., AND BOLES, D., 1995. Dynamic Storage Allocation: A Survey and Critical Review. In *IWMM '95: Proceedings of the International Workshop on Memory Management*, pages 1–116. Springer-Verlag, London, UK. ISBN 3-540-60368-9 (cited on pages 8 and 44)

WILSON, P. R., LAM, M. S., AND MOHER, T. G., 1991. Effective Static-Graph Reorganization to Improve Locality in Garbage-Collected Systems. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 177–191. Toronto, Canada (cited on pages 34 and 46)