Garbage Collection and the Case for High-level Low-level Programming

Daniel Frampton

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

June 2010

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

Daniel Frampton
8 June 2010

to Kerryn.

Acknowledgments

This work would not have been possible without the generous support and assistance of many individuals and organizations. I only hope that I have remembered to give you all the credit you deserve.

First, I would like to thank my advisor, Steve, who has been amazing throughout my candidature. He has provided me with incredible opportunities, encouraging me to apply for internships, supporting me in attending several conferences. Most of all, he has helped me to believe in and persist with my research—thank you.

I would also like to express my gratitude to those who have supported me financially: the Australian National University, the Australian Government, IBM, and Microsoft Research.

During my time as a student I have been lucky to be involved with several great research communities, including DaCapo, Microsoft Research's Singularity and Advanced Compiler Technology groups, and IBM's Metronome group. Each of these has helped to shape my research, and my involvement with them has been both enjoyable and productive. Thanks specifically to Bjarne, Mark, Vivian and David at MSR, to Dave, David, Mike and Perry at IBM, and to Amer, Eliot, Kathryn, Steve, Tony, and the countless others in DaCapo that make it such a friendly and engaging community.

I would also like to thank my fellow students, particularly Robin and Filip, with whom I have worked most closely. I have greatly enjoyed our many intense discussions and coding marathons, and look forward to more of the same in the future!

I would also like to express my gratitude to all the developers of Jikes RVM and MMTk, past and present, without whom this research would not have been the same.

For offering to review previous versions of this document, thanks must go to Kerryn, Steve, Kathryn, Luke, and Mira. The document is all the better for having your attention, though as always, I claim exclusive ownership of any and all remaining errors (as unlikely an occurence as this may be!). Particular thanks must go to Kerryn, who not only spent countless hours reviewing my work, but also helped me to tame Illustrator, and waded with me through countless poorly formatted BibTeX entries to get my bibliography under control.

Last of all, I would like to recognize the vast amount of more personal support that I have received throughout this endeavor. Thanks to my parents, Luke, Mira, Danel, Jackie, all of my soccer buddies, and especially to Kerryn, who has helped to make the whole process of finishing infinitely more enjoyable than it otherwise could have been.

Abstract

Modern high-level programming languages have helped to raise the level of abstraction at which software is written, increasing reliability and security, while also reducing development costs. Despite the benefits afforded by high-level languages, low-level applications—such as real-time applications and systems programs—have bucked the general trend and continue to be overwhelmingly written in low-level languages such as C. Software complexity is continuing to escalate, reliability and security are now first-order concerns, and hardware is increasingly turning to more radical designs to deliver performance increases. In this environment, the use of low-level languages as the rule for low-level programming is becoming increasingly questionable.

These trends raise the question: what is holding back the use of high-level languages for low-level applications? Two key technical barriers are: 1) the limited expressiveness of high-level languages, which often intentionally abstract over detail required to implement needed low-level functionality; and 2) the unique performance requirements of low-level applications, which often demand a combination of throughput, responsiveness, and predictability.

My thesis is that high-level languages can and should be used for low-level applications, improving security and reliability, reducing development cost, and combating increasing hardware complexity.

I have addressed this challenge of high-level low-level programming through: 1) the development of high-performance garbage collection mechanisms and algorithms, in particular those that deliver on the performance requirements of low-level applications; and 2) the use and refinement of a suitably expressive high-level low-level programming approach in the development of garbage collection techniques.

This thesis describes techniques to improve garbage collection performance and introduces two novel garbage collection approaches—*Cycle Tracing* and *Generational Metronome*—that provide the combination of throughput and responsiveness demanded by low-level applications. This thesis presents a framework for high-level low-level programming that provides tools to construct new abstractions around relevant low-level features. This thesis also draws on experience gained through engineering garbage collectors, and shows how visualization can be an invaluable tool in understanding, debugging, and evaluating complex software systems.

These contributions are reinforced through case studies, showing that high-level low-level applications can meet strict performance requirements *while maintaining the benefits in design afforded by high-level languages*. This work demonstrates that high-level low-level programming is both possible and beneficial, leaving the most significant roadblock to adoption a cultural one, not a technical one.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis addresses the problem of writing high-quality low-level software, asking *how we can*—as well as *why we need to*—leverage high-level languages for low-level programming tasks.

1.1 Definitions

We start by defining the use of the relative terms *high-level language* and *low-level programming* in this thesis. We use *high-level languages* to describe languages that provide type-safety, memory-safety, encapsulation, and strong abstractions over hardware. We define *low-level programming* as that which requires transparent, efficient access to the underlying hardware and/or operating system. Low-level programming includes a wide range of applications, but my focus has been on two key applications: 1) modern language runtimes; and 2) real-time systems.

1.2 Problem Statement

The complexity of computer hardware has increased dramatically, evolving from simple in-order processors with uniform memory access to multicore, heterogeneous, out-of-order, super-scalar processors with non-uniform cache hierarchies. The trend of increasing complexity is only set to continue as hardware designers turn to multi- and many-core designs in order to provide further performance increases [Agarwal et al., 2000].

While perhaps the most confronting change is the increasing importance of concurrency, there is also a trend towards heterogeneous systems with special purpose cores. Notable examples include the synergistic processing elements in the Cell [Kahle et al., 2005], the use of graphics processors for more general computation [Garland et al., 2008], and even the dynamic generation of custom processors using FPGAs [Huang et al., 2008]. While the distinction between the cores in many of these examples is quite clear, one can foresee a future in which large groups of heterogeneous processor resources must be managed dynamically by the next generation of systems software.

In addition to changes in hardware design, greater demands have been placed on software in terms of complexity, with larger, more sophisticated software systems, and increased demands for reliability and security. These requirements are extremely challenging for developers of low-level systems software in particular, because low-level software is the foundation on which application software is built; without secure, scalable, and reliable low-level software, there is no hope to satisfy software requirements into the future.

Earlier evolutions of computer hardware and software requirements demanded a transition from assembly programming to languages such as C. As hardware and software complexity continue to increase, application programmers are increasingly choosing high-level languages such as C# and Java which raise the level of abstraction.

Modern high-level languages, which provide strong abstractions over hardware, have helped to improve how software is written; high-level languages insulate programs from changes in hardware, provide concepts such as type- and memory-safety to avoid key classes of programming error, and facilitate building large, secure, and reliable applications in a modular and extensible manner. These high-level programming trends have, for the most part, not yet reached low-level systems software.

There are exceptions, however, and I have been involved with three research projects that seek to benefit from the application of high-level languages to low-level programming tasks: IBM's Jikes RVM[1] (a Java virtual machine written in Java); Microsoft's Singularity[2] (an operating system written in C#); and IBM's Metronome[3] (to allow real-time applications to be written in Java). Each of these projects has helped to shape my research direction.

1.3 Scope and Contributions

Given the potential software engineering advantages of high-level languages, the question one must ask is: "What is preventing the widespread adoption of high-level languages for low-level programming?" The aim of my research has been to *identify roadblocks*, and *develop novel techniques* to resolve them.

There are two sources of resistance to bringing high-level low-level programming into the mainstream: technical and social. I seek only to address the technical issues, and have identified three key areas in which the state of the art must be improved to support the goal of high-level low-level programming:

Expressivity. The key to high-level languages is abstraction, yet this abstraction often defeats one of the primary requirements of low-level programming: transparent access to detail. To enable high-level low-level programming, it is essential for high-level languages to express necessary low-level aspects, *without* breaking the high-level language abstractions and thus negating the advantage of

[1] http://www.jikesrvm.org

[2] http://research.microsoft.com/en-us/projects/singularity/

[3] http://domino.research.ibm.com/comm/research_projects.nsf/pages/metronome.index.html

the language. This thesis describes an approach and a concrete framework, `org.vmmagic`, for high-level low-level programming in Java.

Garbage collector performance. Low-level code places different demands on runtime systems than general application software. While improvements in optimizing compiler technology have largely resolved the issue of absolute code performance, one of the key benefits of high-level languages—*memory safety*—relies on garbage collection, which often yields undesirable performance characteristics in terms of throughput, responsiveness, and predictability. This thesis describes techniques for improving collector performance, as well as two novel garbage collection algorithms that aim to deliver the combination of throughput and responsiveness demanded by low-level applications.

Development tools. The increase in hardware and software complexity has a direct impact on the ability for developers to evaluate, understand, and debug systems. Ultimately, the correctness of a low-level program is judged by observed behavior, however, increased complexity makes it difficult to observe and analyze the behavior of low-level programs. Development tools, in particular those that harness visualization techniques, are well suited to improving this process. This thesis describes *TuningFork*, a visualization platform for debugging and evaluating real-time applications running on a real-time Java virtual machine.

This thesis details contributions in each of these areas, with research driven through the development of novel garbage collection approaches for low-level programs, including experience using a high-level low-level programming approach.

1.4 Thesis Outline

The body of this dissertation is structured around two key elements arising from the areas listed in the previous section: 1) garbage collection techniques and algorithms, and 2) engineering low-level programs in high-level languages. Garbage collection is discussed first as it provides needed context for the chapters that follow.

Chapter 2 provides an overview of garbage collection and surveys relevant garbage collection literature. Chapters 3–5 detail contributions to improving garbage collection approaches for low-level systems. Chapter 3 describes novel techniques to improve overall collection performance by improving the performance of fundamental mechanisms, while Chapters 4 and 5 discuss new garbage collection algorithms. Chapter 4 describes *Cycle Tracing*, a technique based on reference counting that provides throughput and responsiveness in the face of cyclic garbage. Chapter 5 details *Generational Metronome*, a garbage collector that provides improved time and space behavior while maintaining real-time guarantees.

Chapter 6 starts with a brief history of language use for low-level programming, and then describes previous approaches to high-level low-level programming. Chapters 7–9 discuss my contributions to improving the way low-level systems, in par-

ticular garbage collectors, are engineered. Chapter 7 identifies the requirements for high-level low-level programming, then describes a concrete framework to support it built on language extension: `org.vmmagic`. The high-level low-level programming approach is demonstrated in Chapter 8 through a discussion of real-world experience. Then, Chapter 9 shows how visualization can be used to help address the problem of understanding low-level programs in the face of increasing hardware and software complexity.

Finally, Chapter 10 concludes the thesis—describing how my contributions have shown that high-level low-level programming is both beneficial and feasible—and identifies future directions for research.

Chapter 2

Garbage Collection

Garbage collection is an essential component of modern high-level languages, enabling strong type-safety and memory-safety guarantees. However, garbage collection has the potential to adversely affect performance, in terms of *throughput*, *responsiveness*, and *predictability*. This chapter provides an overview of garbage collection, covering fundamental algorithms and mechanisms. The focus is on garbage collection approaches that have the potential to address *all* of these performance criteria simultaneously, yielding predictable, highly-responsive, high-throughput systems.

Section 2.1 provides a brief overview of garbage collection, defines necessary terminology, and introduces fundamental *algorithms* and *mechanisms*. Section 2.2 discusses the performance requirements of low-level programs with respect to garbage collection. Sections 2.3 and 2.4 then describe garbage collection techniques that have the potential to meet the particular performance requirements of low-level programming. Section 2.3 discusses work on *incremental* and *concurrent* tracing garbage collection: techniques that allow garbage collection to proceed alongside application activity. Section 2.4 then describes an alternative approach based on reference counting, an inherently incremental approach to garbage collection often used in low-level programming.

2.1 The Anatomy of a Garbage Collector

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 "Garbage Collection: Algorithms for Automatic Dynamic Memory Management" [Jones and Lins, 1996], and "Uniprocessor Garbage Collection Techniques" [Wilson, 1992].

Programs require data to execute, and this data is typically stored in memory. Memory can be allocated statically (where memory requirements are fixed ahead-of-time), on the stack (tightly binding the lifetime of the data to the currently executing method), or *dynamically*, where memory requirements are determined during execution—potentially changing between individual executions of the same program. This dynamically allocated *heap* memory can be explicitly managed by the program (through primitives such as the C functions `malloc` and `free`), or it can be

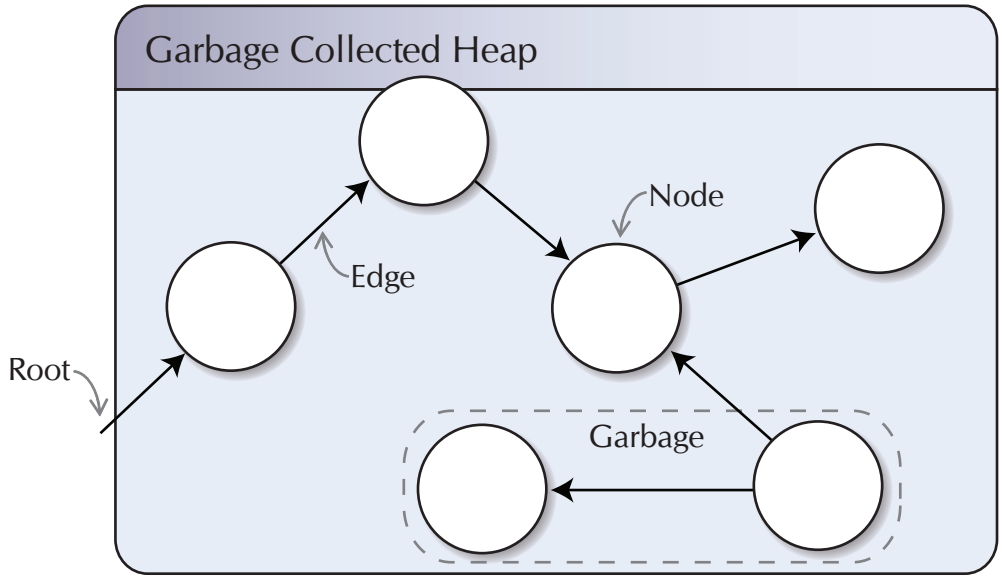


Figure 2.1: An object graph, showing *nodes*, *edges*, and a *root*.

automatically managed through the use of a garbage collector.

Garbage collection takes the burden of explicitly managing memory away from the programmer. While there are many cases in which this burden is insignificant, complex systems with large, shared data-structures make the explicit management of memory both an onerous and error-prone task. The need to manage memory explicitly also compromises software design, forcing additional communication between modules in order to ensure that a *global* consensus is reached before any shared data is freed.

The role of the garbage collector is to reclaim memory that is no longer required by the application. To assist the discussion of garbage collection, we will view all objects in memory as a *directed graph* as shown in Figure 2.1. Objects are represented as *nodes*, and references between objects are represented as directed *edges*. There are also edges originating from outside the object graph—such as values held in program variables—which are known as *roots*. In accordance with the terminology of Dijkstra et al. [1978], application threads that manipulate this object graph (by allocating new objects and changing the set of edges) are known as *mutators*, while threads that perform garbage collection work are known as *collectors*.

Determining precisely when an object will no longer be accessed is difficult in general, so garbage collectors rely on a conservative approximation based on *reachability*. Any object that is *unreachable*—that is, there is no path from a root to the node over the edges in the graph—can never be accessed again by the application, and may therefore be safely reclaimed.

2.1.1 Taxonomy of Garbage Collection Algorithms

Memory management approaches can be categorized based on how they solve three key sub-problems: object *allocation*, garbage *identification*, and garbage *reclamation*. Naturally, approaches to each of these sub-problems have a synergistic relationship with solutions to other sub-problems. Many memory management approaches are hybrids, drawing on several approaches to each of these sub-problems.

2.1.1.1 Object Allocation

There are two fundamental techniques used for object allocation—*bump pointer* allocation, and *free list* allocation.

Bump pointer allocation. Under bump pointer allocation (see Figure 2.2), memory is allocated by running a cursor across memory, *bumping* the cursor by the size of each allocation request. Bump pointer allocation is simple and fast at allocation time, but provides no facility for incremental freeing. Given the simplicity of the approach, the design space for bump pointer allocation is quite restricted. One key design consideration is the approach used to allow parallel allocation. Bump pointer allocation schemes generally perform synchronized allocation of larger chunks [Garthwaite and White, 1998; Alpern et al., 1999; Berger et al., 2000], which are then assigned to a single thread—allowing fast, unsynchronized bump pointer allocation within the chunk.

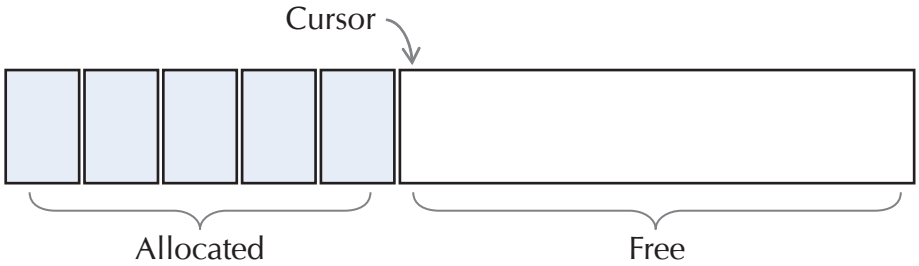


Figure 2.2: Bump pointer allocation.

Free list allocation. Under free list allocation (see Figure 2.3), memory is divided into cells, which are then maintained in a list—the free list. Throughout this thesis, the free list structure of most interest is the *segregated* free list, such as that described by Boehm and Weiser [1988].[1] The segregated free list scheme (described as a *two-level* allocation scheme by Jones and Lins [1996]) attempts to balance the concerns of fragmentation and throughput performance. In a segregated free list scheme an allocator manages *multiple* free lists, each containing a list of empty cells of a single fixed

[1]The design space for free list allocation is large; refer to Jones and Lins [1996] or Wilson et al. [1995] for a more complete discussion.

size. Because each list contains cells of a fixed size, allocation is fast and no searching is required. Memory is divided up into larger *blocks*, each containing cells of a fixed size. Blocks are then managed on a block free list, with empty blocks available for use by *any* size class. This structure addresses fragmentation for most programs, failing only when a program: 1) allocates many objects of a given size class; 2) keeps a small fraction alive (pinning down many blocks); and then 3) changes allocation patterns to allocate many objects of *different* size classes. This pathology is generally rare and can be addressed through some form of copying collection.

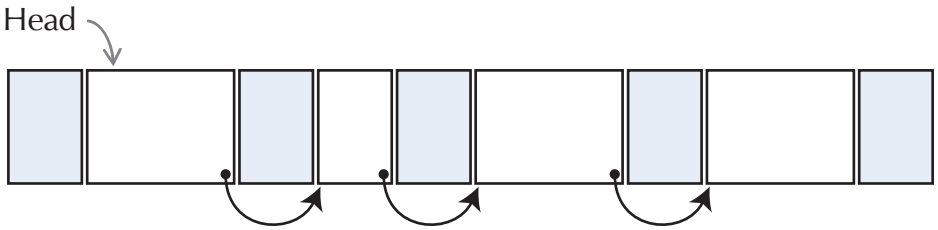


Figure 2.3: Free list allocation.

2.1.1.2 Garbage Identification

There are two fundamental techniques for identifying garbage data: *reference counting* and *tracing*. Each of these techniques forms the basis for one or more of the canonical garbage collection algorithms described in more detail in Section 2.1.2.

Reference counting. This method *directly* identifies garbage. Each object has a *reference count*, which keeps track of the number of references (incoming edges) to that object in the object graph. When a reference count falls to zero, the associated object can be considered garbage.

Tracing. This method *indirectly* identifies garbage by directly identifying all live objects. Tracing involves performing a *transitive closure* across some part of the object graph—visiting all objects transitively reachable from some set of root edges—identifying each visited object as live. All objects that were *not* visited during the trace are identified as garbage.

2.1.1.3 Garbage Reclamation

Once objects have been allocated, and those that need to be collected have been identified, there are several techniques that can be used to reclaim the space.

Direct to free list. For direct garbage collection approaches (e.g., reference counting) it is possible to directly return the space for objects to a free list.

Evacuation. Live objects can be *evacuated* from a region of memory, which once emptied of live data, may be reclaimed in its entirety. This approach requires another region of memory into which the live objects can be copied. Evacuation can be particularly effective when there are very few survivors, and naturally aligns itself with tracing as the approach for identification.

Compaction. Compaction rearranges the memory within the region *in-place* to allow future allocation into the region. A classic example is sliding compaction [Styger, 1967; Abrahams et al., 1966] where all live data is compressed into a contiguous chunk of used memory, leaving a contiguous chunk of memory free for future allocation.

Sweep. A sweep is a traversal over allocated objects in the heap, freeing the space associated with objects that have been identified as garbage. Some form of sweep is required by many tracing approaches, because garbage is not identified directly. While sweep generally operates over individual free list cells, it is also possible to use the sweep approach on larger regions of memory.

2.1.2 Canonical Algorithms

This section briefly introduces canonical algorithms that cover the design space laid out above.

2.1.2.1 Reference Counting

One of the classic forms of garbage collection is reference counting [Collins, 1960]. Recall from above that reference counting works by keeping track of the number of incoming edges—or references—to each node in the object graph. When this count drops to zero, the object is known to be unreachable and may be collected. Figure 2.4 shows an object graph with reference counts calculated, and also demonstrates the fundamental weakness of reference counting: *cyclic garbage*. The objects X and Y are clearly unreachable (there is no path to either of them from any root) but they will never be collected because they still hold counted references to each other. Unless additional work is performed to identify and collect it, cyclic garbage can cause memory leaks.

Reference counting is of particular interest for high-level low-level programming, and is discussed in detail in Section 2.4. It is inherently incremental, and uses object-local information only, rather than requiring any global computation. Explicit reference counting—where the programmer manually manipulates reference count information—is a proven approach for low-level programming, extensively used for managing data structures in low-level software written in languages such as C and C++.

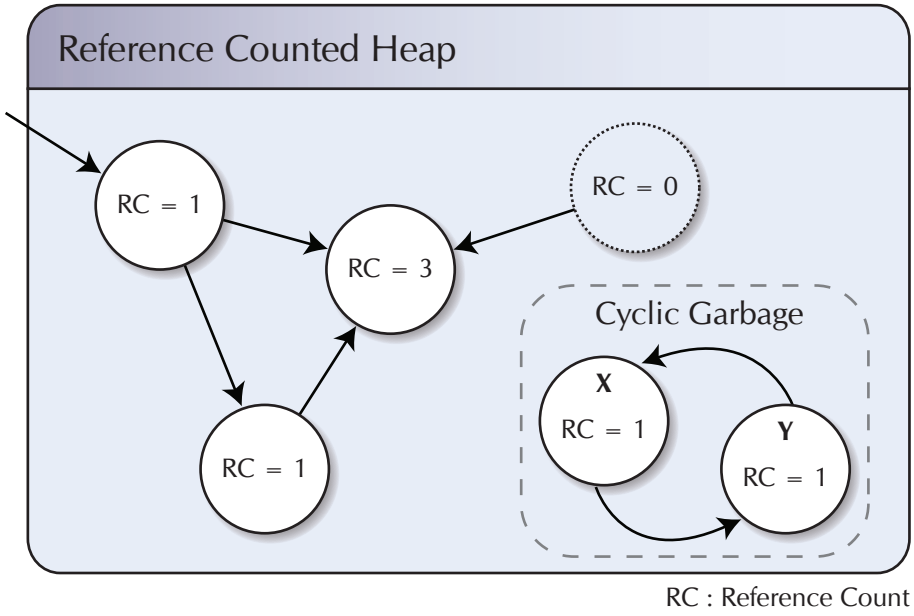


Figure 2.4: An object graph showing *reference counts* and *cyclic garbage*.

2.1.2.2 Mark-Sweep

Mark-sweep collection [McCarthy, 1960] is a *tracing* collection approach that runs in two simple phases:

1. A *mark* phase, which performs a transitive closure over the object graph, *marking* objects as they are visited.

2. A *sweep* phase, where *all* objects in the heap are checked, and any that are not marked are unreachable and may be collected. It is possible for part of this sweeping phase to be performed during execution, a technique called *lazy sweeping* which reduces garbage collector time, and can actually improve *overall* performance, due to the sweep operation and subsequent allocations being performed on the same page, improving cache behavior.

Mark-sweep collection is efficient at collection time, but forces the mutator to allocate objects in the discovered holes surrounding live objects. This can result in lower allocation performance, as well as generally exhibiting poor locality of reference due to objects being spread over the heap. Over time, mark-sweep can also encounter problems with fragmentation—even though the sum total of available memory may be sufficient, an empty, *contiguous* region of sufficient size may not be available.

2.1.2.3 Semi-Space

Semi-space collection is also based on tracing, but uses a very different approach to reclaim memory. Memory is logically divided into two equally sized regions.

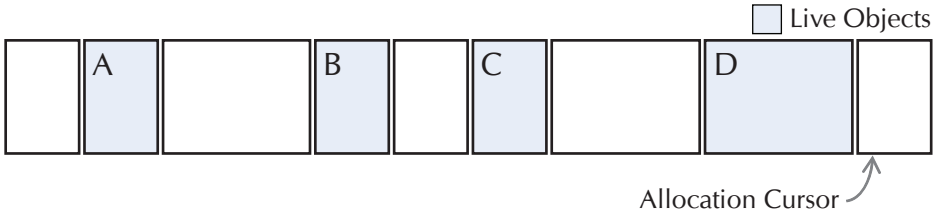
During program execution one region contains objects, while the other region is empty. When garbage collection is triggered, the region containing objects is labeled as the *from-space* and the empty region is labeled as the *to-space*. Garbage collection proceeds by performing a transitive closure over the object graph, copying all nodes encountered in from-space into to-space—updating all edges to point to the copied objects in to-space. At the end of collection all reachable objects have been copied out and saved, and all that remains in the from-space is unused. This from-space is now considered empty, and the to-space contains all live objects: a reversal of the roles of the two regions prior to the collection. Initial implementations of copying collectors used a recursive algorithm [Minsky, 1963; Fenichel and Yochelson, 1969] but a simple iterative algorithm was later introduced by Cheney [1970]. In comparison to mark-sweep collection, semi-space collection:

- makes less efficient use of memory as it must hold 50% percent of total memory as a *copy reserve* to ensure there is space to copy all objects in the worst case;
- can be more expensive at collection time because all live objects must be copied;
- can be *cheaper* at collection time if very few objects survive, because no sweep phase is required;
- can utilize efficient bump pointer allocation, because free memory is always maintained as a contiguous block; and
- has less problems with fragmentation, because live objects are copied into a contiguous chunk of memory.

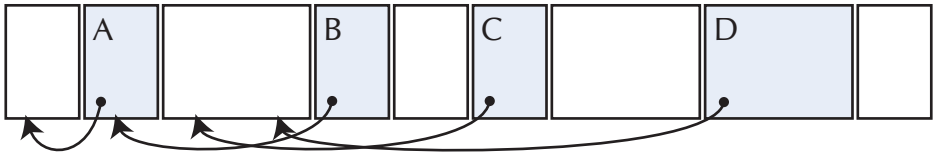
2.1.2.4 Mark-Compact

Mark-compact collection aims to combine the benefits of both semi-space and mark-sweep collection. It addresses fragmentation often seen in mark-sweep by *compacting* objects into contiguous regions of memory, but it does so *in place* rather than relying on the large copy reserve required by semi-space collection. While this in-place transition saves space, it typically involves significant additional collection effort. This is because additional care must be taken to ensure that the target location of a copied object does not contain live data. A simple form is *sliding* compaction, which logically compresses all live objects—in allocation order—into a single contiguous chunk. A simple sliding compaction algorithm—known as the LISP-2 algorithm [Styger, 1967]—proceeds as follows, with the state at the conclusion of key phases shown in Figure 2.5:

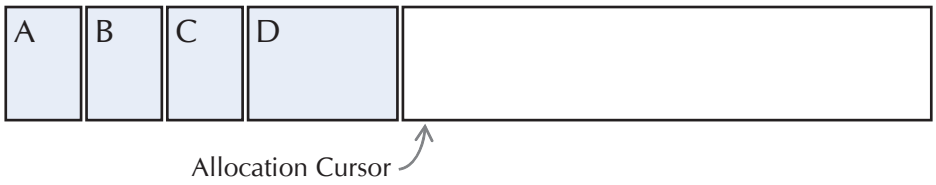
1. A *mark* phase, which performs a transitive closure over the object graph, *marking* objects as they are visited.
2. A *compute-forwarding-pointers* phase, where objects are processed *in address order* and the future location of each marked object is calculated. The calculation occurs by simply incrementing a cursor by the size of each live object as it is encountered.



(a) After the *mark* phase.



(b) After the *compute-forwarding-pointers* phase.



(c) After the *relocate* phase.

Figure 2.5: Sliding compacting collection.

3. A *forwarding* phase, where all pointers are updated to reflect the addresses calculated in the previous phase. Note that after this phase all references point to *future* locations of objects, rather than the *current* location.

4. A *relocation* phase, where objects are copied to their target locations in address order. The address order is important as it ensures that the target location does not contain live data.

The additional phases of simple compaction algorithms make them significantly more expensive than simple mark-sweep or semi-space collection. While there have been many attempts to reduce this cost—by optimizing the phases, reducing the number of phases, or by implementing the phases as operations on compact representations of the heap [Kermany and Petrank, 2006]—compaction is rarely used as the sole collection strategy in a high-performance system. Compaction is, however, commonly combined with mark-sweep collection (to provide a means to escape fragmentation issues), and is often used alongside semi-space collection [Sansom, 1991] to allow execution to continue when memory is tight.

2.1.2.5 Mark-Region

Mark-Region is a collection approach that combines contiguous allocation and non-copying tracing collection. The motivation of this approach is to combine the mutator performance of semi-space with the collection performance of mark-sweep. In terms of allocation, mark-region is similar to semi-space, with objects allocated into contiguous *regions* of memory using a bump pointer. In terms of collection, mark-region is similar to mark-sweep, but sweeps entire *regions*; regions with no reachable objects are made available again for contiguous allocation. Immix [Blackburn and McKinley, 2008] provides the first detailed analysis and description of a mark-region collector, although a mark-region approach was previously used in JRockit [Oracle] and IBM [Borman, 2002] production virtual machines. A mark-region collection proceeds as follows:

1. A *mark* phase, which performs a transitive closure over the object graph, marking *regions* which contain live objects as each object is visited.

2. A *sweep* phase, where *regions* that were not marked in the previous phase are made available for future contiguous allocation.

The mark-region approach is susceptible to issues with fragmentation, because it may not be possible to discover large contiguous blocks to allow efficient bump pointer allocation. To combat this, mark-region collectors often employ techniques to relocate objects in memory to reduce fragmentation. The JRockit collector performs compaction of a fraction of the heap at each collection, the IBM collector performs whole heap compaction when necessary, and Immix performs lightweight defragmentation as required when memory is in demand.

2.1.3 Generational Collection

Generational garbage collection [Lieberman and Hewitt, 1983; Moon, 1984; Ungar, 1984] is perhaps the single most important advance in garbage collection since the first collectors were developed in the early 1960s. The generational hypothesis states that most objects have very short lifetimes. Generational collectors are optimized for when this hypothesis holds, and thereby attain greater collection efficiency by focusing collection effort on the most recently allocated objects.

Generational collectors partition the heap into *generations* based on allocation age. This thesis considers only the basic form of generational collection, where the heap is divided into two generations: the *nursery*—containing the most recently allocated set of objects—and the *mature* area—containing all other objects. In order to independently collect the nursery, generational collectors must remember all pointers from the mature space into the nursery. This can be achieved by building a *remembered set* of pointers created into the nursery, or by remembering regions of the mature space (usually referred to as *cards*) that contain nursery pointers, and must be scanned at nursery collection time. References from the mature space into the nursery, in combination with any other roots (e.g., program variables), then provide the starting point for a transitive closure across all live objects within the nursery. A partial copying collection can be performed during this closure, with live objects evacuated from the nursery into the mature space. When the generational hypothesis holds, this collection is very efficient because only a small fraction of nursery objects must be copied into the mature area.

2.1.4 Barriers

Barriers are operations that are injected into mutator code surrounding mutator operations that may affect the garbage collector. Barriers are most commonly inserted on read and write operations, and are essential tools for more powerful garbage collection algorithms. In reference counting collectors, reference write barriers can be used to perform necessary reference count increments and decrements. Generational collectors may also rely on reference write barriers to intercept pointers created from mature objects to nursery objects. Concurrent and incremental garbage collectors may make extensive use of read and write barriers to keep them informed of potentially destructive changes, and to ensure that mutators are operating on and updating the appropriate data.

Given the algorithmic power of barriers, it is essential that high-performance barrier operations be available in order to judge the true cost of a given garbage collection approach. The performance of barriers is a complex interaction of multiple factors including the operation itself, how it is optimized (e.g., what portions are inlined), the behavior of individual applications, and the underlying architecture [Hosking et al., 1992; Blackburn and McKinley, 2002; Blackburn and Hosking, 2004]. It is also possible to elide barriers to improve performance by identifying cases in which the barrier operation is redundant [Vechev and Bacon, 2004].

2.2 Garbage Collection for Low-level Programs

Garbage collection can deliver significant software engineering benefits, but the presence of garbage collection can adversely affect application performance. To achieve the requisite combination of throughput and responsiveness demanded by low-level programs, it is necessary to use garbage collection techniques that allow both collectors and mutators to progress within relatively short time windows. This section discusses how garbage collection can affect mutator performance, and introduces metrics to help quantify the problem. Sections 2.3 and 2.4 then describe existing garbage collection strategies that allow mutator progress during a collection cycle.

Given that the collector operates over the same object graph as mutators, there is the potential for conflict between mutator and collector. In order to safely and accurately perform collection work, collection algorithms are thus forced to either stop all mutators from proceeding during collection work, impose some synchronization burden onto both collector and mutator, or a combination of both. This has significant potential to affect mutator performance, both directly—in terms of reduced processor time and collector synchronization overheads—as well as indirectly—through changes in mutator locality (positive or negative) and changes in timing behavior due to unpredictable collector interference. Overall throughput performance can suffer if too much processor time is used for garbage collection, or if the choice of collection and/or allocation technique negatively affects data locality. Predictability can be affected because the amount of time required to perform seemingly identical operations can change (even by orders of magnitude) when garbage collection is active. Some activities cannot tolerate significant variation, and so need garbage collectors that provide responsiveness guarantees. Strong responsiveness guarantees typically come with a significant cost to overall throughput, and while this is acceptable for some applications, it is generally not acceptable for low-level programs.

2.2.1 Measuring Garbage Collection Interference

Given that low-level programs may have strict performance requirements in terms of throughput and responsiveness, there is a clear need to both *specify* what these requirements are, as well as identify a means to *evaluate* the characteristics of individual approaches.

2.2.1.1 Throughput

The throughput, or overall execution time required to perform a given task, can be affected by the presence of garbage collection. This effect can be both direct (the collector itself requires resources to run) and indirect, with the choice of collector changing how mutator code is executed.

Fine-grained effects—due to barriers, changes in allocation policy, changes in cache locality, or small increments of collection work—can impose a penalty on the mutator that is difficult to measure dynamically. For practical purposes, collectors designed to guarantee responsiveness must aim to make the burden imposed by the

collector predictable, by making it either constant (i.e., independent of the current phase of the collector) or a simple function of the current collector phase (so that it can be compensated for in any calculations). This thesis assumes the former model, where any performance guarantees, along with any mutator time measurements, are inclusive of any garbage collector burden. This approach is suitable for the collectors discussed in this thesis, in which most operations have a fairly uniform cost. The approach would, however, be problematic for work-based collectors such as Baker's collector [Henry G. Baker, 1978], where the cost of individual operations can vary greatly due to the collection state of objects.

In order to compare mutator throughput, total mutator time can be measured for each system across a single workload. This comparison accounts for primary effects (such as the choice of allocator and barrier operations) as well as secondary effects (such as changes in cache locality).

2.2.1.2 Responsiveness and Predictability

Some garbage collectors interrupt the execution of mutators by taking exclusive access to the heap. While this simplifies the synchronization of mutator and collector activity, it has the potential to affect the responsiveness of the application, which is a critical concern for many systems. With responsiveness as a key requirement, it is important to have a metric to measure it. The length of the longest interruption—or maximum pause time—that is imposed on the mutator by a garbage collector is an obvious choice, but maximum pause time can be a poor measure of overall responsiveness. While a mutator cannot make progress during a long collector pause, it is also possible that a mutator may not make *sufficient* progress if it encounters a sequence of many *short* pauses.

Looking at the problem from the perspective of an application, suppose we have a unit of application work that is guaranteed to take at most 7ms to complete (assuming the application is given all resources), and that the application is required to complete such a unit of work within 10ms. To guarantee that this is possible (given that the application may execute this unit of work at any moment) we must ensure that for any given 10ms of execution time at least 7ms is given to the mutator. While it is impossible to guarantee such behavior if there are individual collection pauses longer than 3ms, more than three 1ms collection pauses in an individual 10ms window would also not meet the requirement. In order to guarantee that the 7ms application work unit can execute within an arbitrary 10ms window, we can restate our requirement to say that the application must have a *minimum mutator utilization* (MMU) [Cheng and Blelloch, 2001] of 70% within each 10ms time window.

Note that a 70% per 10ms MMU also bounds the overall amount of time consumed by the garbage collector across program execution to 30%. The ability to guarantee a higher MMU is generally easier over longer time periods, where the impact of individual pauses can be amortized, allowing the specification of several MMUs at different window sizes (e.g., 50% per 2ms, 70% per 10ms, and 80% per 100ms). Conversely, it becomes increasingly difficult to provide an MMU guarantee

as the time window becomes smaller. In the limit, it becomes necessary to introduce additional programming environments that can safely run concurrently to the collector, and seamlessly integrate into the high-level language environment [Bollella and Gosling, 2000; Spoonhower et al., 2006; Titzer, 2006; Auerbach et al., 2007b; Spring et al., 2007; Auerbach et al., 2008b].

While minimum mutator utilization handles the key situation in which measuring pause time is problematic—bursts of many short pauses—it is by no means a perfect metric for measuring responsiveness, or assessing the suitability of a collection strategy for real-time systems. In real-time systems it is often possible to use a *slack-based* scheduling approach where garbage collection increments are scheduled *around* real-time tasks. The correctness of such a system is governed by there being sufficient slack surrounding real-time tasks to execute garbage collection work, although the garbage collector itself may not meet any MMU guarantees. MMU provides an indication of overall responsiveness, and is well suited to many classes of real-time applications, and makes it possible to develop a model to guarantee real-time behavior with only a few high-level application and collector characteristics. MMU and slack based scheduling can be combined in a hybrid approach, such as the Metronome Tax-and-Spend [Auerbach et al., 2008a] collector, which meets an MMU specification but uses slack to accumulate a surplus of collector work to allow longer periods of sustained mutator utilization.

2.3 Concurrent and Incremental Tracing Collection

Given strict requirements for responsiveness, an obvious goal is to allow mutator progress *during* a collection cycle. There are two broad approaches to allowing mutator progress during tracing collection: *incremental* collection, and *concurrent* collection. This thesis uses the term *incremental* collection to describe systems where a single collection cycle is broken into *increments* of collector work, but where only mutators *or* collectors are running at any point in time. *Concurrent* collection is used to describe systems where both collector and mutator threads are running and progressing simultaneously.

Under both concurrent and incremental collection, some form of synchronization between collector and mutator must be performed to ensure that: 1) all garbage is ultimately collected, 2) no live memory is incorrectly collected, and 3) garbage collection is guaranteed to terminate. There is a substantial literature on concurrent garbage collection with seminal work dating back to the 1970s [Steele, 1975; Dijkstra et al., 1978].

2.3.1 The Tricolor Abstraction

To facilitate the discussion of concurrent and incremental collection, this section uses the tricolor abstraction of Dijkstra et al. [1978], where each node in the object graph is colored according to collection status:

White nodes have not been visited by the collector. All nodes start white, and those that remain white at the end of collection are garbage.

Gray nodes have been visited by the collector, but have not yet had their *outgoing* edges inspected.

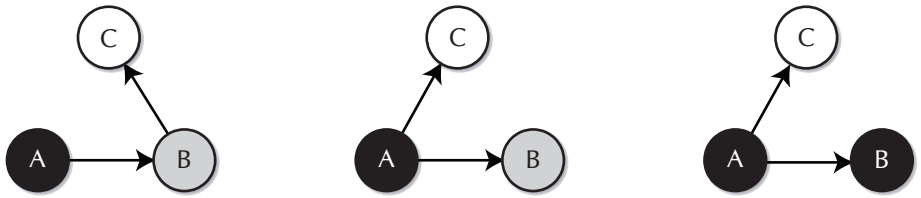
Black nodes have been visited by the collector, and all outgoing edges have been traversed.

2.3.2 The Mutator–Collector Race

Figure 2.6 illustrates the fundamental race that an incremental or concurrent tracing collector must address. At t_0, the collector has processed A, marking the sole referent B gray (enqueuing it for later processing), and finally marking A black. At t_1, the mutator has created an edge from A to C, and removed the edge from B to C. At time t_2 the collector resumes execution by processing B, which involves simply marking B as black as there are no outgoing edges from B. C is thus left white at the end of the collection, but it is still live. Jones and Lins [1996, Chapter 8] state the conditions for this race to cause the collection of live data to be:

C1. A pointer from a black object to a white object is created.

C2. The original reference to the white object is destroyed.



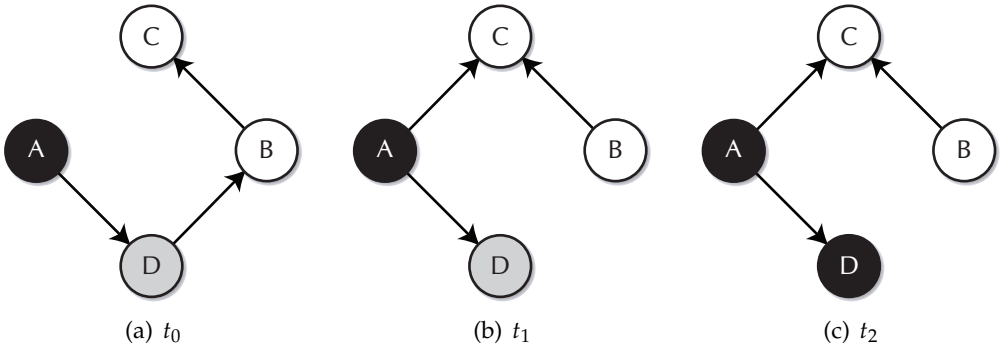
(a) t_0 (b) t_1 (c) t_2

Figure 2.6: The mutator–collector race.

However, as Figure 2.7 shows, the introduction of an intervening node, D, between A and B makes it clear that C2 must be modified to account for the white object becoming *indirectly* unreachable. The following modified condition accounts for this case:

C2′. The original *path* to the white object is destroyed.

The following sections provide a brief overview of the many previously described concurrent collection algorithms, focusing in particular on how they attempt to handle this race condition. The design space of concurrent collection is broad. There have been studies of the relationship between the various non-moving concurrent approaches [Vechev et al., 2005, 2006], but these do not extend to include approaches that move objects.



(a) t_0 (b) t_1 (c) t_2

Figure 2.7: Adding an extra node to the mutator–collector race.

2.3.2.1 Incremental-Update Algorithms

Incremental-update algorithms use write barriers (see Section 2.1.4) to monitor mutations to the heap and directly ensure that no pointers from black to white objects are created. There are two seminal incremental update algorithms, both dating from the mid 1970s. Dijkstra et al. [1978] describe a conservative approach, marking the *target* of a new reference gray if it was white, regardless of the color of the source. Steele [1975] describe a less conservative approach, marking the *source* gray if it was black and the target was white. Abstractly, Steele's algorithm retreats the gray wave-front, while Dijkstra et al.'s advances it. Conservatism in this case affects the volume of *floating garbage*: garbage that will not be reclaimed by the current collection cycle and will consume memory until after the *next* garbage cycle.

Extension to support multiple mutators. The Doligez-Leroy-Gonthier algorithm [Doligez and Leroy, 1993; Doligez and Gonthier, 1994] is an extension of Dijkstra et al.'s algorithm to work in the case of multiple mutator threads, and has been subsequently implemented for Java [Domani et al., 2000b,a]. The key idea is to use a *handshake* mechanism to allow mutators to agree on phase changes without requiring all mutators to stop. Handshake mechanisms are now frequently used in concurrent collectors to perform operations such as enabling or disabling barriers, flushing or clearing data structures, executing special machine instructions, or changing the color to be used for newly allocated objects.

2.3.2.2 Snapshot-at-the-Beginning Algorithms

Snapshot-at-the-beginning algorithms aim to collect the heap as it exists at the start of the collection by taking a logical snapshot. In practice, it is too expensive to snapshot the whole heap, so practical algorithms detect potentially destructive changes to the heap, and then process or copy those parts of the heap before the destructive change occurs. Yuasa [1990] achieves this by using a write barrier which inspects the value of the field *before* it is updated, and if this value points to a white object, the white object is made gray. Azatchi et al. [2003] improve on this algorithm by using a lightweight

object remembering barrier. The first time an object is mutated during each collection, the original state of every reference in the object is *logged*, with a pointer to the log recorded in an additional word in the object's header. Any unlogged objects encountered during collection are also logged prior to processing. All of these logged copies of objects make up a virtual *before-image* of the state of all references as at the start of the collection, which is what is traced by the collector. All snapshot-at-the-beginning approaches may create significant amounts of floating garbage, because they make no attempt to collect any garbage created after the snapshot.

2.3.3 Incremental and Concurrent Copying

While good allocator design may reduce the problem of fragmentation [Johnstone and Wilson, 1998], when attempting to provide strict bounds—on time, space, or utilization—the issue of worst case (rather than common case) fragmentation must be addressed.[2]

The previous two sections describe techniques to ensure that the correct set of objects were identified as garbage. In the case of either incremental or concurrent copying collection, there is the additional race between the collector relocating an object, and the mutators performing reads and writes of the fields of the object. Section 2.3.3.1 discusses the problem of incremental copying, essentially described as a problem of ensuring that mutators operate on the correct version of an object. Concurrent copying—far more complicated as mutations are allowed to occur *during* the copy operation of an object—is discussed in Section 2.3.3.2.

2.3.3.1 Incremental Copying Algorithms

Henry G. Baker [1978] introduced an incremental version of the simple semi-space algorithm (see Section 2.1.2.3) for uniprocessors, where small increments of work are performed at each allocation. Because copying progresses incrementally, the system must ensure that mutators see a consistent view of the heap. The simplest method to achieve this—as used in Baker's work—is with a reference read barrier (code that is executed whenever the mutator reads a reference) to maintain a *to-space invariant*. This invariant ensures that the mutator can only ever access objects that have been fully processed by the collector. When a reference into from-space is encountered, the mutator must perform an additional increment of collection work (and either copy the object or forward the reference if copying has already been performed). Brooks [1984] modifies Baker's algorithm by using a read barrier that unconditionally indirects to the current version of the object, trading off additional space to store a forwarding pointer in every object against a reduction in code size and improved run-time performance.

[2] It is possible to entirely avoid external fragmentation by supporting only a single object size [Siebert, 2000; Baker, 1992]. This is not a general solution because it increases internal fragmentation and also imposes a significant run-time performance hit due to additional indirections and calculations when accessing data.

Appel et al. [1988] avoid the use of a read barrier altogether by leveraging hardware protection mechanisms for virtual memory to detect accesses to from-space (a refinement due to Johnson [1992] improves this by reducing the per-page scanning work required). Ben-Yitzhak et al. [2002] and Kermany and Petrank [2006] also use memory protection based approaches to allow partial compaction during mostly non-moving collection. Click et al. [2005] introduce a similar approach but use custom hardware combined with a special mutator read instruction that traps to the collector when the target reference is on a page that is being relocated.

Nettles and O'Toole [1993] require neither a read barrier nor virtual memory hardware protection. Mutators continue to operate on from-space objects (instead of to-space objects) during collection, and use a write barrier on *all* fields (including non-reference fields) to maintain a *mutation log*. This mutation log is then used by the collector to create a consistent copy of the heap.[3] Meyer [2006] implements the read barrier for a Baker-style collector in custom hardware, which is extended by Stanchina and Meyer [2007] into a hardware-based mostly non-copying collector of similar design.

Eager versus lazy barriers. It is interesting to note that to-space or from-space invariants can be enforced *eagerly* or *lazily* [Bacon et al., 2003b] through barrier operations. *Eager barriers* ensure that all references in registers or on the stack meet the invariant, meaning that the barrier is applied to references being brought into the set (e.g., when a reference is read from the heap). *Lazy barriers* do not enforce an invariant on references held in registers on the stack, but instead ensure that each *use* of a reference is protected to ensure the invariant is met (e.g., just prior to a reference value being used for any heap operation). For naive implementations there appears to be a clear trade-off: lazy barriers allow more incremental processing (because the stack and registers can be lazily updated), but are likely to do more work because work must be done for *every* heap access, not just reference accesses. However, as stated by Bacon et al. [2003b], when subjected to aggressive optimization, both operations tend towards a common solution, and are therefore similar in terms of cost.

2.3.3.2 Concurrent Copying Algorithms

The copying algorithms described in the previous section do not perform the copying operation concurrently: mutators are either all suspended while copies are made, or stalled if they try to access an object that is being copied. Concurrent mutation and copying of individual objects adds significant complexity. Herlihy and Moss [1992] work around this problem by making objects immutable and maintaining modifications as a singly linked list of *object versions*. A change to an object field thus involves making a copy, updating field values, and then atomically installing the link to the

[3] These mutation log techniques were implemented in the context of functional programming languages (ML), and—due to the race between mutations and the collector—are likely to be less suited to imperative or object-oriented languages which tend to have far higher mutation rates and fewer immutable objects.

new version. If this fails it indicates that there was a race to update and it is necessary to update to the new latest version and retry. Although valuable from a theoretical standpoint, it is hard to see how this algorithm could be implemented efficiently in practice.

Sapphire [Hudson and Moss, 2001] performs copying collection concurrently in Java without the use of a read barrier, by using a write barrier to *replicate* mutations (of reference and non-reference fields) in older versions of objects to the in-progress copy. Sapphire may provide unexpected results when concurrent updates to an object field occur. Two updates u_1 and u_2 to a single field f could be committed to memory in the order u_1u_2 on the old version of the object, but replicated as u_2u_1 on the new version of the object, making it possible for the observed value of the field f to change when the switch from the old version to the new version occurs. While the Java memory model permits this behavior (due to the lack of synchronization) the effect may be surprising to programmers so is generally considered undesirable.

Stopless [Pizlo et al., 2007] provides concurrent copying while maintaining a more understandable memory model than Sapphire through the use of a *wide* object during copying, where each field has a corresponding status word for synchronization. Fields are individually and atomically copied into the wide object, using an atomic compare-and-swap operation of the field alongside its corresponding status word. Once in the wide object, a second phase of copying returns the object to a normal, narrow object format, again using atomic compare-and-swap operations on the wide object to ensure correctness. Staccato [McCloskey et al., 2008] and Chicken [Pizlo et al., 2008] avoid the need for a wide object by optimistically performing the copy but invalidating and abandoning the copy when mutation occurs by marking the header of each object during each update. Pizlo et al. [2008] also describe Clover, a modification of Stopless that uses a randomly chosen marker value, α, to indicate that an individual field has been forwarded. Mutators use compare-and-swap operations to write to fields, and when they detect α values they read from the new copy of the object. The downside of this approach is that any mutator wanting to write an α value must wait for copying to complete, although a good choice of α would make this quite unlikely. Clover thus trades an absolute guarantee that the mutator will not have to wait for an improvement in execution time.

2.4 Reference Counting Collection

This section discusses reference counting, an alternative to the concurrent and incremental tracing collectors discussed above, that also allows collector activity to proceed alongside mutators. Reference counting, as an inherently incremental approach to collection, is a natural fit for applications that require high levels of responsiveness.

Collins [1960] first described reference counting garbage collection. As described in Section 2.1.2.1, simple reference counting approaches work by maintaining the count of all incoming edges for each object. Two obvious shortcomings of reference counting are the cost required to maintain such counts, and the inability for refer-

ence counting to collect self-referential data structures. These two broad issues are discussed in Sections 2.4.1 and 2.4.2 respectively.

2.4.1 Reducing Performance Overheads

Naive implementations of reference counting require increment and decrement operations to be performed for almost every reference operation, including operations on local variables. This substantial cost is further increased on multithreaded systems, because reference count updates and reference mutations must be performed atomically to ensure correct counts are maintained. The following sections discuss approaches that aim to reduce the overhead required for maintaining reference counts.

2.4.1.1 Compiler Optimization

There is an obvious potential for using compiler analysis to reduce the amount of reference counting work required [Barth, 1977; Joisha, 2006, 2007]. Such approaches make it possible for only the *net effect* of a sequence of code to be processed on the heap, avoiding unnecessary increment and decrement operations. Take, for example, the case that a reference count of an object is incremented and then immediately decremented; the two updates can *cancel* with no actual update required.

2.4.1.2 Deferred Reference Counting

Deutsch and Bobrow [1976] introduced *deferred* reference counting. In contrast to the immediate reference counting scheme described above, deferred reference counting schemes *ignore* mutations to frequently modified variables—such as those stored in registers and on the stack. Periodically, these references are enumerated into a *root set*, and any objects that are neither in the root set nor referenced by other objects in the heap may be collected. Deutsch and Bobrow achieve this directly by maintaining a *zero count table* that holds all objects known to have a reference count of zero. This zero count table is enumerated, and any object that does not have a corresponding entry in the root set is identified as garbage. Bacon et al. [2001] avoid the need to maintain a zero count table by *buffering decrements* between collections. At collection time, elements in the root set are given a *temporary increment* while processing all of the buffered decrements.

2.4.1.3 Coalescing Reference Counting

Levanoni and Petrank [2001, 2006] observed that all but the first and last in any chain of mutations to a given pointer could be coalesced. Only the initial and final states of the pointer are necessary to calculate correct reference counts—the intervening mutations generate increments and decrements that cancel each other out. Consider a pointer field f that first points to object A. If it is mutated to point to B and later C, the increment and decrement to B cancel each other out, and the reference counts of only A (decremented by 1) and C (incremented by 1) need to be modified.

This observation can be exploited by remembering only the initial value of a reference field between periodic reference counting collections. At each of these collections, only the objects referred to by the initial (stored) and current values of the reference field need be updated.

2.4.1.4 Ulterior Reference Counting

Azatchi and Petrank [2003] and Blackburn and McKinley [2003] concurrently and independently added generations to reference counting. Blackburn and McKinley [2003] did so in a general framework of *ulterior reference counting*, which generalizes the notion of deferral to include heap pointers—such as those within a copying nursery. Using this configuration, Blackburn and McKinley showed that it was possible to achieve performance competitive to the fastest tracing collectors while also exhibiting reduced pause times. However, these results were achieved only for benchmarks that did not show a significant volume of cyclic garbage.

2.4.2 Collecting Cyclic Garbage

Jones and Lins [1996] give a good description of the various approaches that deal with cyclic data structures. Most approaches are not general, being either language specific, or dependent on programmer intervention. However, there exist two general approaches: backup tracing [Deutsch and Bobrow, 1976] and trial deletion [Christopher, 1984; Martínez et al., 1990; Lins, 1992; Bacon and Rajan, 2001; Bacon et al., 2001].

2.4.2.1 Backup Tracing

Backup tracing performs a mark-sweep style trace of the entire heap to eliminate cyclic garbage. Unless tracing is performed concurrently, reference counting's advantages of prompt reclamation and low pause times are lost. However, as described in Section 2.3, concurrent mark-sweep collection imposes an additional synchronization burden on the mutator. In the case that a significant fraction of garbage is cyclic garbage, a reference counting system with backup tracing essentially reverts to a mark-sweep collector with additional overheads in both time and space for maintaining reference counts.

2.4.2.2 Trial Deletion

Intuitively, trial deletion algorithms determine whether the only references to a data structure originate within the same data structure, thereby forming a garbage cycle. The original trial deletion algorithm can be traced back to Christopher [1984], who described a method to implement complete garbage collection without requiring access to an external root set.

Trial deletion algorithms work by selecting some candidate object or objects, and then performing a partial mark-sweep [Jones and Lins, 1996] of the object graph.

If it can be determined that a candidate is alive only by virtue of reachability from itself, then it is part of a self-sustaining garbage cycle and should be collected. The process of *trialing* the deletion of these candidate objects proceeds in three phases, with each phase performing a transitive closure over the sub-graph reachable from the candidates:

1. Traverse the sub-graph, adjusting reference counts on all objects in the sub-graph to reflect the hypothetical death of the candidates. At the end of this phase the reference counts reflect only the references from objects external to the sub-graph. Any object with a count of zero is reachable only from within the sub-graph.

2. Traverse the sub-graph, incrementing the reference counts of all objects pointed to by externally reachable objects—that is the objects whose count did *not* drop to zero in the trial.

3. Traverse the sub-graph, sweeping any objects that still have a zero count.

The original implementation by Christopher [1984] effectively applied this three-phase approach using *all* objects in the heap as the candidate set. Martínez et al. [1990] noted that cyclic garbage can be created only when a reference is removed from an object that had multiple incoming references. Lins [1992] noted that this could be prohibitively expensive, and performed cycle detection lazily, periodically targeting the set of candidate objects whose counts experienced decrements to non-zero values. Bacon and Rajan [2001] made three key improvements to Lins' algorithm: 1) they performed each phase over all candidates *en masse* after observing that performing the three phases for each candidate sequentially could exhibit quadratic complexity; 2) They used very simple static analysis to exclude the processing of objects they could identify as inherently acyclic and 3) they extended the algorithm to allow it to run concurrently with the mutator.

2.5 Real-Time Collection

The previous sections described collection strategies that allow mutator progress during a garbage collection cycle. This section discusses work on systems with proven bounds on performance in terms of time, space, or utilization. The following paragraphs give an overview of key previous work on real-time collection, and then Section 2.5.1 gives a more complete description of Metronome, a state-of-the-art real-time collector that forms the basis for my work in Chapter 5.

Strong guarantees in terms of time, space, and utilization can be important for many application areas, and establishing such bounds helps us to better understand how different algorithms behave. As indicated in the previous section, reference counting collection alone is not complete, due to the possibility of cyclic garbage. For this reason, proven bounds on general purpose reference counting collectors would require guarantees on the cycle collector, a strictly harder problem than providing

similar guarantees on a tracing collector. For that reason, the published literature on real-time collection that this section discusses is all based on incremental and/or concurrent tracing collection.

Henry G. Baker [1978] introduced the first collector to provide guarantees on time and space. Baker's work was restricted to uniprocessor systems, however, and while many multiprocessor algorithms were implemented and published in the intervening time, Blelloch and Cheng [1999] were the first to describe a multiprocessor collector with proven bounds on time and space. Cheng and Blelloch [2001] describe a more practical implementation of this multiprocessor algorithm, and also introduce the Minimum Mutator Utilization Metric (see Section 2.2.1.2), but do not provide proven bounds on utilization.

2.5.1 Metronome

Metronome, described by Bacon et al. [2003b], was the first collector to provide a model that gave time, space, and utilization guarantees. This model, refined by [Bacon et al., 2003a], requires the quantification of only a few key characteristics of the application and collector, such as allocation and tracing rates. Metronome is a hard real-time incremental collector. It uses a hybrid of non-copying mark-sweep collection (in the common case) and selective copying collection (when fragmentation occurs).

The virtual machine scheduler alternates between execution of mutator threads and garbage collector threads, using predictable quanta and predictable spacing between those quanta.

A key contribution of the Metronome system is that it abandons a fine grained work-based approach—such as that of Henry G. Baker [1978]—in favor of a *time-based* approach. The fundamental observation here is that the race between collector and mutator occurs at a relatively coarse temporal granularity—a collection cycle. Bursty allocation behavior in small time windows can then be amortized over the period of a complete collection cycle. A time-based scheduler interleaves mutator and collector work in small quanta at a ratio determined by the model. This ensures that the collector keeps up, but does so in a predictable manner amenable to providing guarantees on mutator utilization.

Metronome is a snapshot-at-the-beginning algorithm [Yuasa, 1990] that allocates objects black (marked). While this means that no objects allocated after the snapshot can be collected, potentially increase floating garbage, the worst-case performance is no different from other approaches and the termination condition is deterministic—a crucial property for real-time collection.

Metronome addresses fragmentation through several measures. First, large arrays are broken into fixed-size pieces called arraylets, bounding the external fragmentation caused by large objects. Second, Metronome uses a segregated free list (see Section 2.1.1.1), a structure which generally exhibits low fragmentation. Finally, Metronome allows for objects to be relocated during collection. However, fragmentation is rare in practice, so objects are usually not moved. If a page becomes frag-

mented due to garbage collection, its objects are moved to another (mostly full) page containing objects of the same size. Relocation of objects is achieved by using a forwarding pointer located in the header of each object [Brooks, 1984]. A read barrier maintains a to-space invariant (mutators always see objects in the to-space). During marking, all pointers to objects that were relocated in the previous collection are updated, ensuring that at the end of a marking phase the relocated objects of the previous collection can be freed.

Metronome achieves guaranteed real-time behavior provided the application is correctly characterized by the user. In particular, the user must be able to specify the maximum volume of simultaneously live data m as well as the peak allocation rate over the time interval of a garbage collection $a(\Delta G)$. The collector is parametrized by its tracing rate R. Given these characteristics of the mutator and the collector, the user then has the ability to tune the performance of the system using three inter-related parameters: total memory consumption s, minimum guaranteed CPU utilization u, and the resolution at which the utilization is calculated, Δt.

2.6 Summary

This chapter gave a general background of garbage collection, including an overview of key garbage collection mechanisms and algorithms. In particular, the chapter introduced broad approaches to garbage collection that attempt to combine high throughput and responsiveness. The key characteristic of these approaches is the ability for both mutator and collector to progress at the same time. The following chapters build on this work, and introduce novel algorithms that make garbage collection suitable across a wider range of applications, including real-time and low-level applications that have strict requirements in both throughput and responsiveness.

Chapter 3

High-Performance Garbage Collection

Garbage collection plays a central role in modern high-level language runtimes; it is the key technique used to enforce type- and memory safety. However, garbage collection requires resources to fulfill this role, making garbage collection performance an important aspect of overall system performance. This chapter discusses research to improve the performance of garbage collectors through techniques that are largely independent of any single collection algorithm. While performance is of critical concern to many low-level programs, the approaches described in this chapter have broader application beyond the goal of high-level low-level programming.

This chapter describes two projects aimed at improving garbage collector performance. First, Section 3.1 describes how important locality is to the performance of the *transitive closure*—a key mechanism used heavily in garbage collection—and demonstrates how performance can be improved through the use of cache prefetch instructions. Then, Section 3.2 switches focus to a technique to reduce garbage collector load by promptly reclaiming space through a combination of a simple compiler analysis and runtime mechanism.

Section 3.1 is based on work published in the paper "Effective Prefetch for Mark-Sweep Garbage Collection" [Garner, Blackburn, and Frampton, 2007], which discusses both a rigorous analysis of the costs in the tracing loop, and the implementation and evaluation of the effective prefetch system. This chapter focuses on the aspects of the paper for which I was principally responsible, namely those related to the design of the final effective prefetch system.

Section 3.2 discusses work published in the paper "Free-Me: A Static Analysis for Automatic Individual Object Reclamation" [Guyer, McKinley, and Frampton, 2006]. This section focuses on the aspects of the work that I was principally responsible for, namely the selection and implementation of the runtime mechanisms. The section does not discuss the details of the compiler analysis.

3.1 Effective Prefetch for Garbage Collection

Garbage collection typically exhibits poor locality; traversing all objects in the heap can result in a significant number of cache misses. Because the transitive closure process is generally supported by a work queue, it is natural to try and take advantage of cache prefetch instructions to improve performance. Despite this apparently obvious match, previous studies have struggled to provide significant improvements to tracing performance with prefetch instructions. This section shows that effective use of prefetch instructions for tracing is possible by combining prefetch instructions with a restructured tracing loop based on *edge enqueuing*. This configuration allows prefetching to improve the performance of a canonical mark-sweep garbage collector by an average of between 13% and 25% depending on architecture.

This section is structured as follows. First, Section 3.1.1 discusses other work related to prefetching for garbage collection. The two key techniques that together form the basis of the approach—edge enqueuing and buffered prefetch—are discussed in Sections 3.1.2 and 3.1.3 respectively. Finally, Section 3.1.4 provides an overview of the results.

3.1.1 Related Work

While performing a transitive closure, garbage collectors need to maintain some form of work set. The standard approach is to maintain a set of all objects to which an incoming reference has been found, but from which outgoing references have yet to be scanned. Each live object is inserted into the set only once. The tracing loop performs `processNode()`, as shown in Figure 3.1.

```
1 void processNode(Object o) {
2   for(Object e: scan(o)) {
3     if(!isMarked(e)) {
4       mark(e);
5       push(e);
6     }
7   }
8 }
```

Figure 3.1: Core of tracing loop with *node* enqueuing.

Boehm [2000] was the first to apply software prefetching to garbage collection, introducing *prefetch on gray*, a strategy where each object is prefetched upon insertion into the work set (line 5 in Figure 3.2). Boehm reported speedups of up to 17% in synthetic garbage collection dominated benchmarks and 8% on a real application, ghostscript.

Cher et al. [2004] built on Boehm's investigation, using simulation to measure costs and explore the effects of prefetching. They found that when evaluated across a broad range of benchmarks, Boehm's prefetch on gray strategy attained only lim-

ited speedups under simulation, and no noticeable speedups on contemporary hardware. Cher et al. introduce the buffered prefetch strategy that we also adopt (see Section 3.1.3).

Cher et al. validated their simulated results using a PowerPC 970 (G5). They obtained significant speedups on benchmarks from the JOlden suite, but less impressive results for the SPECjvm98 suite, with their best result being 8% on jess, and 2% on javac. All results were achieved using very space-constrained heaps (about 1.125× the minimum heap size) forcing frequent collections, thereby amplifying the effect of any garbage collector performance improvements.

3.1.2 Edge Enqueuing

The typical arrangement of queuing operations in the previous section minimizes queuing operations because each object is enqueued only once. However, we performed analysis that clearly showed that locality, not queuing operations, was the limiting factor in tracing performance. Looking carefully at Figure 3.1, it is clear that data related to each live object must be consulted multiple times: once on line 2 when the object is being scanned, and again on lines 3 and 4 when the mark state is checked and changed. We can reorder the structure of the tracing loop, so that the checking of the mark state is co-located with the scanning of the object. In the case of an object reachable only from one edge, this ensures that all accesses to that object are temporally clustered. For edge enqueuing, the work set contains *edges*, and the tracing loop performs `processEdge()` as shown in Figure 3.2 for each edge.

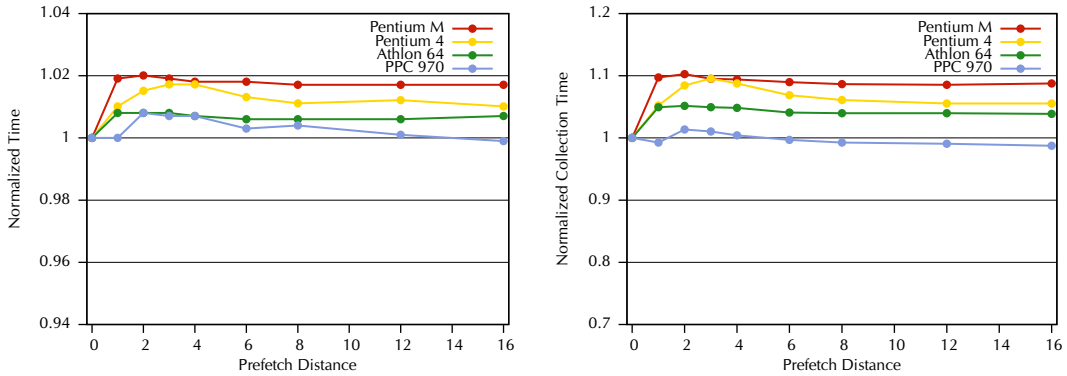
```
1  void processEdge(Object o) {
2    if(!isMarked(o)) {
3      mark(o);
4      for(Object e: scan(o)) {
5        push(e);
6      }
7    }
8  }
```

Figure 3.2: Core of tracing loop with *edge* enqueuing.

Edge enqueuing requires at least as many queuing operations as node enqueuing, as each live node in the object graph must have *at least* one edge to it. When edge enqueuing was previously considered (e.g., Jones and Lins [1996]) it was discarded on the basis of this overhead. However, on modern hardware *locality* is the key performance concern during tracing, so it is important to consider the potential for edge enqueuing to align memory accesses and improve cache behavior.

3.1.3 Buffered Prefetch

The poor locality exhibited by the tracing loop and the presence of a work set makes the use of prefetch instructions a natural choice. In order to allow prefetch, however,



(a) Total Time (b) Collection Time

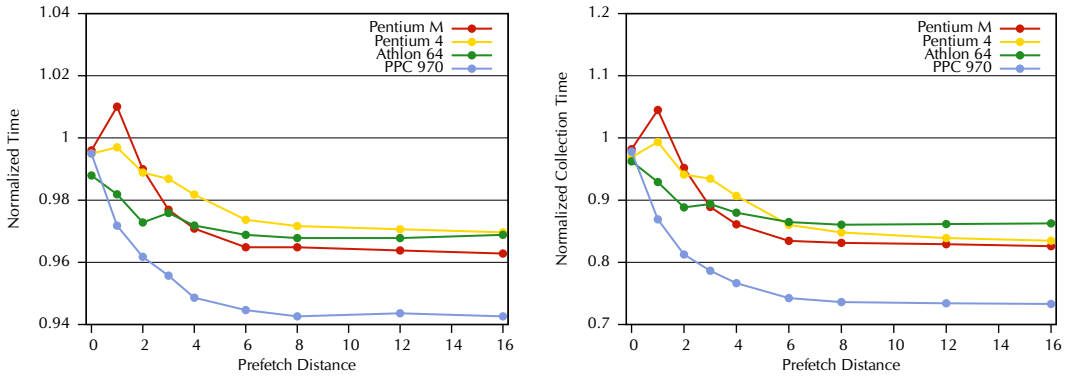
Figure 3.3: Performance for *node* enqueuing across architectures and prefetch distances.

it is important to inject the necessary prefetch call into the instruction stream at the correct point. Initial attempts at using prefetch simply executed a prefetch instruction for each object as it was inserted onto a LIFO work stack [Boehm, 2000]. This does not provide significant gains for real world applications as the distance between the prefetch and the actual read is highly unpredictable. Often the prefetch is executed too late and occurs just before the object data is used—providing no time for the processor to prefetch the data into cache—and in other cases the prefetch happens too early, meaning that the data is loaded into and subsequently replaced in the cache before it is required.

In order to address these issues Cher et al. [2004] introduced the idea of a buffered prefetch, where a small fixed-size FIFO buffer is used in combination with the main LIFO stack. New values are pushed directly to the stack, while values are processed off the FIFO buffer. When the latest value is taken off the FIFO buffer, another value is popped off the stack, inserted into the buffer, and a prefetch instruction executed for that value. The prefetch distance can be easily varied by changing the size of the FIFO buffer. This FIFO buffer in combination with the main LIFO stack exhibits mostly LIFO behavior, but provides a more predictable distance between the prefetch and memory accesses. Our approach uses a similar queue structure, but as discussed in the previous section, we combine it with an edge enqueuing approach.

3.1.4 Results

We evaluated the effectiveness of software prefetching in the tracing loop using both edge and node enqueuing models. Figures 3.3 and 3.4 show performance for node and edge enqueuing models respectively, providing total and collection time performance for prefetch distances from 0 to 16 across four modern architectures. Each graph shows the geometric mean of performance for the full set of 17 benchmarks drawn from DaCapo [Blackburn et al., 2006], SPECjvm98 [SPEC, 1999], and the pjbb2000 variant of jbb2000 [SPEC, 2001]. Results are normalized to the performance



(a) Total Time (b) Collection Time

Figure 3.4: Performance for *edge* enqueuing across architectures and prefetch distances.

of node enqueuing with no prefetch: the original default configuration.

We report results for a fairly generous heap, 3× the minimum heap size for each benchmark. We performed identical experiments across a range of heap sizes, and found the collection time improvements were independent of heap size. Naturally, overall improvements to total time increased because the total fraction of time spent in collection was higher in smaller heaps. These results show that combining edge enqueuing with reasonable prefetch distances can significantly improve collection performance, yielding a 14% to 27% average speedup across all benchmarks (depending on architecture). This translates into a 3% to 6% improvement in total execution time.

3.2 Free-Me: Prompt Reclamation of Garbage

Garbage collection is well understood to trade off time and space [Blackburn et al., 2004a; Hirzel et al., 2002; Inoue et al., 2003; Shaham et al., 2001; Ungar, 1984]. Without performing garbage collection, space requirements are unbounded. Conversely, performing extremely frequent garbage collection minimizes space requirements. Free-Me improves the balance between these two extremes with an approach that promptly reclaims *some* memory *without invoking garbage collection*. This reduces space consumption while avoiding the need to spend time running the garbage collector frequently. This is achieved through a novel combination of a lightweight compiler analysis and runtime mechanism. The analysis identifies points in the program where memory can be reclaimed, and injects calls to the runtime mechanism.

The key insight in Free-Me is that it is not necessary to try and cover all cases; Free-Me's analysis is not required to identify and free all objects, nor is the runtime mechanism required to immediately recycle any memory identified as garbage. This allows the overheads of Free-Me—both at compilation time and execution time—to be kept low, ensuring that the approach is profitable.

Free-Me adds an explicit `free()` operation to the garbage collector, and uses a

Free-Me compiler analysis to inject calls to `free()` at the point an object becomes garbage. Free-Me analysis combines simple, flow-insensitive pointer analysis with flow-sensitive liveness information. Even when an allocation site produces some objects that escape the method or loop, Free-Me can still free those that do not escape. Since its scope is mostly local, it typically finds very short-lived objects. Free-Me includes an interprocedural component that summarizes connectivity and identifies *factory* methods (simple methods that return a newly allocated object and have no other side effects).

The underlying garbage collector dictates the implementation of `free()`. This work describes the implementation of `free()` for free list and bump pointer allocation. For bump pointer allocation, several variants of `free()` were explored: 1) a version that can reclaim the object only when it is the most recent allocation, requiring strict last-in-first-out ordering of calls to `free()`; 2) a more powerful version of `free()` that tracks one unreclaimed region closest to the bump pointer, allowing a subsequent call to `free()` to coalesce with it and free a larger region; and 3) a version that simply reduces the required copy reserve size by the number of free bytes, logically saving only half as much space but doing so with minimal complexity.

3.2.1 Related Work

While Free-Me allocates all objects on the heap, the approach is related to region allocation [Chin et al., 2004; Hicks et al., 2004; Qian and Hendren, 2002; Tofte and Talpin, 1997] and stack allocation [Blanchet, 2003; Choi et al., 2003; Gay and Steensgaard, 2000; Whaley and Rinard, 1999], although it differs in two key ways. First, region and stack allocation require lifetimes to be contained within a particular program scope, whereas our approach frees objects as soon as they become unreachable. Second, region and stack allocation require specialized allocation sites; the choice of stack, region, or heap allocation must be made when the object is allocated. In many systems, this limitation requires each allocation site to produce objects with the same lifetime characteristics. Even if some objects become unreachable, these systems must wait until all objects become unreachable.

Research on understanding object lifetimes in Java programs has provided motivation for this work. Marinov and O'Callahan [2003] show that using *object equivalence* (in which object contents are the same, but have disjoint lifetimes) could save 2% to 50% of memory across SPECjvm98 and two Java server programs. Inoue et al. [2003] explore the limits of lifetime predictability for allocation sites, and find that many objects have zero lifetimes, indicating that an approach such as Free-Me could be profitable.

3.2.2 Runtime Mechanism

We implemented and evaluated several mechanisms for Free-Me across a range of collectors. The following sections discuss four distinct implementations of `free()`. Each `free()` implementation takes two arguments: a reference to the object that is

no longer live, and the size of the object in bytes. The size information is generally available to the compiler as a constant, but when the size is not known statically (e.g., for arrays with dynamic length), the size is calculated by querying type information.

3.2.2.1 Free List Implementation

Our free list implementation is suitable for several collectors, including mark-sweep, reference counting, and variants of mark-compact. This implementation can take advantage of a more aggressive compiler analysis, because both long and short-lived objects can be freed.

The base free list is a segregated free list, where free lists of fixed sized cells are maintained. The point in time at which cells can be identified as free varies depending on the collection policy. The implementation of `free()` is unaffected by these variations as it can always link the freed cell onto the head of the free list for the given size class, making the cell available for immediate re-allocation.

3.2.2.2 Bump Pointer Implementations

Bump pointer allocation is a very simple form of allocation and is widely used with copying, compacting, and generational garbage collectors. Allocation simply proceeds in memory order, bumping a pointer to accommodate newly allocated objects. Two variants of `free()` for bump pointer allocation were evaluated: `unbump()` and `unbumpRegion()`. In addition, we evaluated a variant, `unreserve()`, that does not actually free space, but instead allows a reduction in the required copy reserve.

`unbump()` The simplest implementation decrements the bump pointer when the most recently allocated object becomes garbage. Subsequent allocations can then reuse the space that was occupied by the object. If an attempt is made to free an object that was not the most recently allocated, then `unbump()` has no effect.

`unbumpRegion()` One of the complexities of the `unbump()` approach is that the compiler is forced to present the calls in last-in first-out order in order to free multiple objects. `unbumpRegion()` allows the reclamation of more than just the most recently allocated object, by keeping track of a single contiguous region of freed data just before the bump pointer. If a sequence of calls to `unbumpRegion()` results in the region growing to the current bump pointer, then the entire contiguous region is made available for allocation.

`unreserve()` A copying garbage collector is required to maintain a *copy-reserve* which contains sufficient space to allow all surviving objects to be copied into it. `unreserve()` uses the identification of some objects as garbage which therefore cannot survive to reduce the required copy-reserve, improving space efficiency while requiring very little in terms of overhead.

3.2.3 Results

Free-Me was implemented in Jikes RVM and MMTk, a high performance Java-in-Java virtual machine and memory management toolkit [Alpern et al., 1999, 2000; Blackburn et al., 2004a,b]. This section presents two sets of results. First, Section 3.2.3.1 presents statistics of the effectiveness of the analysis in injecting calls to `free()` across the various benchmarks and configurations. Then, Section 3.2.3.2 presents performance numbers—for total, garbage collector, and mutator time—across the various configurations.

3.2.3.1 Effectiveness of Analysis

Table 3.1 presents statistics for our compiler analysis gathered using a specially instrumented build. On average, the Free-Me analysis frees 32% of all objects and up to 80% in one benchmark. Table 3.1 also shows results for two other systems that are more restrictive than Free-Me. *Unconditional* uses a modified version of the analysis that requires objects to be dead on all paths. *Stack-like* is even more restrictive, requiring both the frees and allocation to be in the same method, with the call to free restricted to be at the end of the method. These restrictions reduce the average effectiveness from 32% to 25% and 21% respectively, including significant reductions in some benchmarks. Comparing results listed for *Unconditional* and *Free-Me* demonstrates the importance of allowing `free()` on some paths and not others, with the effect most pronounced for more complex benchmarks: bytes freed is reduced by half or more for javac, jack, antlr, bloat, and pmd.

Table 3.2 shows further potential for the refinement of our approach on three benchmarks by modifying the code to make up for shortcomings in the analysis. Note that these modified benchmarks are not used for the performance analysis in the following section. The modifications made are described in the following paragraphs.

javac *inline* A single method, `symbolTable.lookup()`, is hand-inlined as the optimizing compiler chooses not to inline it under normal circumstances. With this change, Free-Me is able to free a total of 27% of objects. An enhanced version of the analysis could change inlining heuristics to work well with Free-Me and automatically detect this case.

db *mod* **and xalan** *mod* The addition of three explicit calls to `free()` from key routines responsible for growing array-based collections. For example, the `ArrayList` container increases the size of its array to accommodate new elements. The `add()` method allocates a new, larger array and copies the elements from the old array. The old array is immediately garbage. Because the arrays are internal to the collection classes, it seems feasible to construct a compiler analysis that can detect and exploit such opportunities.

Table 3.1: Effectiveness of Free-Me analysis, showing total allocation and the percentage of objects that could be freed by Free-Me and two more restrictive approaches.

Benchmark	**Allocated** (MB)	**Percentage Freed**		
		Free-Me	Unconditional	Stack-like
compress	105	0%	0%	0%
jess	263	6%	6%	6%
raytrace	91	81%	80%	80%
db	74	61%	61%	61%
javac	183	13%	9%	9%
mtrt	98	75%	75%	74%
jack	271	60%	47%	38%
pjbb2000	180	19%	9%	3%
antlr	1544	44%	22%	10%
bloat	716	31%	7%	5%
fop	103	30%	24%	20%
hsqldb	515	11%	7%	6%
jython	348	22%	20%	1%
pmd	822	34%	17%	7%
ps	523	4%	4%	3%
xalan	8195	20%	20%	19%
Arithmetic Mean		**32%**	**25%**	**21%**

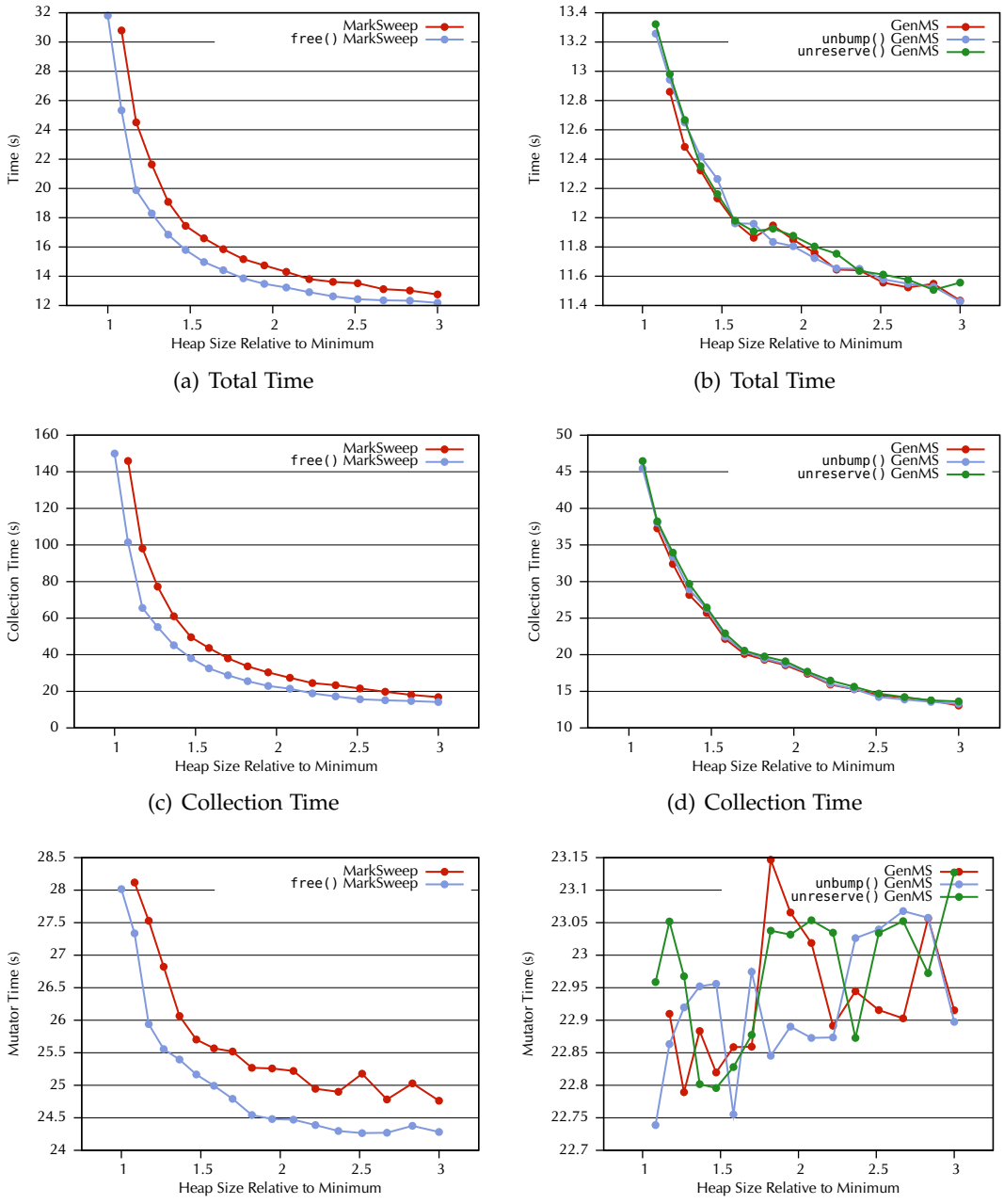
Table 3.2: Effectiveness of Free-Me analysis for hand-modified versions of three benchmarks, showing further potential for the Free-Me approach. Performance for original versions are shown in italics for comparison.

Benchmark	**Allocated** (MB)	**Percentage Freed**		
		Free-Me	Unconditional	Stack-like
db $_{mod}$	74 *(74)*	88% *(61%)*	88% *(61%)*	87% *(61%)*
javac $_{inline}$	188 *(183)*	27% *(13%)*	14% *(9%)*	14% *(9%)*
xalan $_{mod}$	8195 *(8195)*	89% *(20%)*	89% *(20%)*	88% *(19%)*

3.2.3.2 Performance Evaluation

The performance of each Free-Me system was evaluated using SPECjvm98 [SPEC, 1999], SPECjbb2000 [SPEC, 2001], and DaCapo [Blackburn et al., 2006] benchmarks.

Mark-sweep performance. The left half of Figure 3.5 shows that for mark-sweep collection, Free-Me improves total performance by an average of 50% in small heaps, 10% in moderate heaps, and 5% in large heaps. Free-Me improves performance in line with the statistics of freed objects given in Table 3.1. Figure 3.5 shows these improvements are primarily due to a reduction in the time required for garbage collection for the benchmarks. Interestingly, Free-Me also provides a measurable improvement to mutator time, despite the overhead of calling `free()`. There are



(a) Total Time (b) Total Time

(c) Collection Time (d) Collection Time

(e) Mutator Time (f) Mutator Time

Figure 3.5: Total, garbage collection, and mutator time for mark-sweep (left) and generational mark-sweep (right) systems using Free-Me.

two key reasons for this. First, Free-Me reuses the same cell that was recently allocated into, which will exhibit excellent locality. Second, the mark-sweep collector on which Free-Me is built uses lazy allocation, which means that the sweep operation is performed during mutator time. Reducing the number of collections reduces the

number of sweep operations, directly affecting mutator performance.

Generational performance. Free-Me was unable to improve the performance of a state-of-the art generational mark-sweep collector. Although Free-Me reclaims similar numbers of objects, and these frees translate into many fewer nursery collections (e.g., from 16 to 4 in the db benchmark) the survival rate of each nursery collection increases. The cost of collecting the nursery is dominated by the number of objects *copied*, so the reduction in the *number* of nursery collections does not translate into a reduction in overall collection time. Figure 3.5 shows results for two of the variations of `free()` discussed above, `unbump()` and `unreserve()`; `unbumpRegion()` is not shown because it performs strictly worse than `unbump()`.

These results show that generational collectors are extremely effective at reclaiming short-lived objects, making it difficult for any technique (including ours) to outperform it for collecting short-lived objects. However, full-heap collectors are still used in some areas (e.g., real-time systems and memory constrained embedded systems) and Free-Me is a demonstrated way to improve space efficiency and performance while retaining the software engineering benefits of garbage collection.

3.3 Summary

The use of high-performance techniques for garbage collection is an important aspect of overall garbage collection performance. This chapter described two projects that improved the performance of garbage collected systems that have applications across a range of garbage collection algorithms. This is an important avenue for research complementary to the algorithmic approaches that are described in the following chapters.

Chapter 4

Cycle Tracing

Reference counting, introduced in Chapter 2, is an approach to garbage collection that has the potential to deliver the requisite combination of throughput and responsiveness demanded by low-level programs. Much of the recent progress in reference counting has been in reducing the overheads imposed by maintaining object reference counts. While this overhead has been dramatically reduced, the efficient collection of cyclic garbage remains a stumbling block. This chapter addresses this shortcoming with *cycle tracing*, a concurrent, tracing-based cycle collection approach that achieves greater efficiency than existing methods by taking advantage of information already captured and available within a reference counted environment.

This chapter is structured as follows. Section 4.1 describes the cycle tracing algorithm, including a discussion of the base backup tracing algorithm on which cycle tracing is built. Section 4.2 then describes our evaluation methodology and experimental platform, and provides an evaluation of cycle tracing against trial deletion and conventional backup tracing.

The work described in this chapter is presented in "Cycle Tracing: Efficient Concurrent Cyclic Garbage Collection" [Frampton, Blackburn, Quinane, and Zigman, 2009b].

4.1 The Cycle Tracing Algorithm

For programs that create little cyclic garbage, generational reference counters have closed the gap in terms of performance with traditional high-throughput tracing collectors [Blackburn and McKinley, 2003]. Such collectors are able to achieve this while also providing improved responsiveness due to the incremental nature of reference counting collection. The stumbling block is cycle collection. Programs that create even moderate amounts of cyclic garbage suffer severe throughput penalties.

Cycle collection algorithms must determine a transitive property of the object graph. Participation in a cycle depends on the references of an object *transitively* reaching back to the object: a property that can not be determined by looking at an object in isolation. Trial deletion employs partial tracing techniques and tends to have very high overheads for programs with significant volumes of cyclic garbage. High performance cycle collection generally relies on the the use of a *backup tracing* collec-

tor, invoked when insufficient memory is reclaimed by the reference counter alone. While significant research effort has been spent investigating concurrent garbage collection algorithms, surprisingly little research has been undertaken on how best to deploy such collectors within a reference counting environment.

Cycle tracing is a modified version of a snapshot-at-the-beginning concurrent mark-sweep collector. It is designed to collect *only* cyclic garbage, and is optimized for collecting this cyclic garbage in a reference counted environment. Collecting only cyclic garbage is sufficient—all acyclic garbage will be promptly reclaimed by the reference counter. This study identifies and evaluates three potential optimizations over a conventional snapshot-at-the-beginning collector:

1. Taking advantage of information already gathered during reference counting to reduce the set of objects that might have been subject to the collector–mutator race inherent in concurrent tracing collection.
2. Limiting the mark phase to avoid tracing inherently acyclic objects and data structures.
3. Limiting the sweep phase to sweep only potentially cyclic garbage rather than the entire heap.

The following sections first give an overview of the unoptimized base backup tracing algorithm, and then proceed to describe each of these optimizations in detail.

4.1.1 Base Backup Tracing Algorithm

The cycle tracing algorithm is built on a conventional snapshot-at-the-beginning tracing collector as described in Section 2.3.2.2. Abstractly, the backup tracing algorithm can be described in terms of three phases and a single work queue:

1. **Roots.** All objects referenced by roots are added to the work queue.
2. **Mark.** The work queue is exhaustively processed.
   (a) **Process.** Each object is taken off the work queue, and if the object is not marked, it is marked and then each of its referents are added to the work queue.
   (b) **Check.** Before leaving the mark phase, we process any object potentially subject to the collector–mutator race. If any of these objects are not marked, they are marked, added to the work queue, and the Mark phase is resumed.
3. **Sweep.** Reclaim space used by objects that have not been marked.

In the backup tracing algorithm, every object that is modified during the collection must be processed as part of Step 2(b). Throughout the rest of this chapter we refer to this algorithm as the *base snapshot-at-the-beginning algorithm* or simply the *base algorithm*.

4.1.2 A Lightweight Snapshot Write Barrier

This section describes the write barrier used by the backup tracing and cycle tracing systems to support both coalescing reference counting *and* concurrent tracing. Recall from Chapter 2 that a snapshot-at-the-beginning collector relies on a barrier to capture reference values *before* they are mutated, constructing an accurate snapshot of the heap to be traced during collection. Our implementation uses a low-cost, low-synchronization snapshot barrier to provide the required information for both coalescing reference counting *and* concurrent tracing. The barrier is the same as that used by Blackburn and McKinley [2003] which was itself a variant of the barrier used by Levanoni and Petrank [2001] for their on-the-fly reference counter with backup mark-sweep.

```
1  @Inline
2  public void writeBarrier(ObjectReference srcObj,
3                           Address srcSlot,
4                           ObjectReference tgtObj) {
5    if (getLogState(srcObj) != LOGGED)
6      writeBarrierSlow(srcObj);
7    srcSlot.store(tgtObj);
8  }
9
10 @NoInline
11 private void writeBarrierSlow(ObjectReference srcObj) {
12   if (attemptToLog(srcObj)) {
13     enumeratePointersToSnapshotBuffer(srcObj);
14     modifiedObjectBuffer.push(srcObj);
15     setLogState(srcObj, LOGGED);
16   }
17 }
18
19 @Inline
20 private boolean attemptToLog(ObjectReference object) {
21   int oldState;
22   do {      /* perform conditional store */
23     oldState = object.prepare();
24     if (oldState == LOGGED) return false;
25   } while (oldState == BEING_LOGGED ||
26            !object.attempt(oldState, BEING_LOGGED));
27   return true;
28 }
```

Figure 4.1: A low-overhead write barrier to support coalescing reference counting.

Figure 4.1 shows source code for the write barrier. The barrier establishes *before* and *after* images of the pointers within each mutated object, and ensures that each object is only ever remembered once. It achieves this by:

(a) taking a snapshot of the state of all pointer fields of each object prior to its first mutation since a collection (line 13); and

(b) recording the mutated object so that the 'after' state of its fields can be enumerated at collection time (line 14).

Line 5 determines whether the object being mutated has already been logged. This requires a simple *unsynchronized* test of a bit in the object header. If the object fails this test (and appears not to have been logged since the last collection) an out-of-line call is made to the write barrier slow path (lines 10–17). The slow path first attempts to set the state of the object to *being logged*, using a synchronized `prepare/attempt` idiom to ensure the object is only logged once. If the current thread is confirmed as responsible for logging the object, each reference field is processed (lines 13–14), and the object header is then marked as logged (line 15). All subsequent mutations of the object may then fall straight through to line 7 on the fast path and simply perform the (unsynchronized) pointer store.

The common case thus involves only a simple unsynchronized test of a bit in the source object's header in addition to performing the pointer store (lines 5 and 7). The unsynchronized check is sufficient because each object's logged state changes only from unlogged *to* logged in any given collection cycle, and the entire system is guaranteed to have a consistent view of every object's state (all unlogged) at the start of each collection cycle. If any thread is late to observe that an object has been logged it will simply force a call to the slow path where the correct state will be discovered.

Note that this barrier is different to a barrier designed to support a snapshot-at-the-beginning tracing collector *without* reference counting. Line 14 is needed specifically for reference counting; it allows the collector to enumerate *new* reference values within all modified objects at the end of the collection cycle (in order to apply increments). Depending on collector design and architecture, the guard on line 12 may also be omitted, because a snapshot-at-the-beginning tracing collector requires the original value be processed *at least* once, while reference counting must decrement the reference count of each original, overwritten value *exactly* once.

The following sections outline each of the three optimizations cycle tracing applies to the base backup tracing algorithm.

4.1.3 Concurrency Optimization

The first optimization uses reference counting information to simplify the work required to ensure an accurate trace in the face of concurrent mutator and collector activity. Recall that the base backup tracing algorithm must process *every* overwritten reference, because in general, each could be an occurrence of the mutator–collector race and result in live data remaining unmarked. The set of objects that the collector must process to avoid harmful races is known as the *fix-up* set. This section will show that processing every overwritten reference is conservative, because a harmful race will occur only in the case that the *last* reference to a given object is overwritten. In a reference counting environment this information is on hand, so it makes sense to utilize it, thus reducing the size of the fix-up set.

We start by recalling the necessary conditions C1 and C2′ for a race, described in Section 2.3.2 and repeated here:

C1. A pointer from a black object to a white object is created.

C2′. The original path to the white object is destroyed.

In the context of reference counting we make the following claims:

1. For C2′ to occur, *either* the white object, or some object in the original path to the white object, will be subject to a reference count decrement to a non-zero reference count.

2. When C2′ arises, it is correct and sufficient to add to the fix-up set either the white object or any object in the original path still connected to the white object.

We justify the first claim as follows. For case C2′ to occur, some pointer must be removed, and this pointer must:

(a) directly point to the white object; or

(b) be the start of an acyclic path to the white object; or

(c) be the start of a path to the white object that contains a cycle.

Considering case (a), the white object is subject to the following sequence of changes:

1. The initial reference count must be at least 1, because it includes the reference that is to be removed.

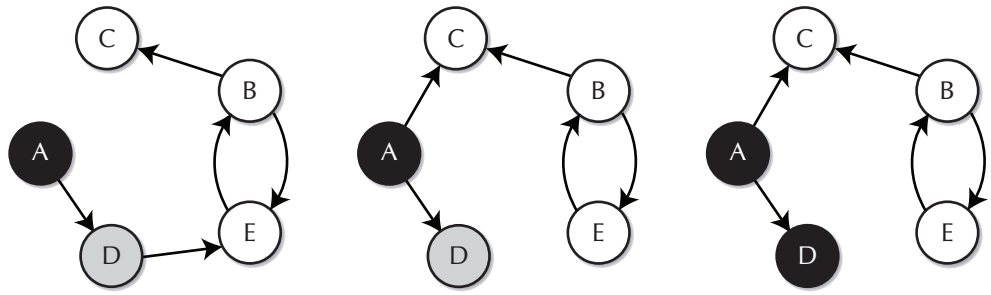
2. During some time interval, the creation of the new pointer (due to condition C1) and the removal of the original pointer to the white object will occur, causing an increment and decrement to the white object to be buffered.

3. The reference count will then rise to at least 2, because buffered increments must be processed before decrements.

4. The reference count will then be subject to a decrement to a non-zero value, as required by our claim.

Case (b) trivially reduces to case (a), because intervening objects are decremented to zero and collected, with referents recursively decremented until the direct pointer to the object is found.

Case (c) is illustrated by Figure 4.2. The deletion of the reference from D to E must cause some object in the path from A to the white object C to have its reference count reduced (i.e., object E), and because E is part of a cycle, this decrement must be to a non-zero reference count, meaning E will be added to the fix-up set. Thus whenever condition C2′ arises, either the white object—or some object in the path to the white



(a) t_0 (b) t_1 (c) t_2

Figure 4.2: Adding a cycle to the mutator–collector race.

object—must experience a decrement to a non-zero reference count, causing it to be added to our fix-up set, satisfying our first claim.

Our second claim is trivial. Since any object in the fix-up set will be traced, it is sufficient to add any object that reaches the white object to the fix-up set. Furthermore, once an object which forms the original path to the white object is made unreachable, the path cannot be changed by the mutator. We therefore claim that it is sufficient and correct to use the set of objects which experienced a decrement to a non-zero reference count as the fix-up set.

Martínez et al. [1990] noted that a decrement to a non-zero reference count is a necessary condition for the generation of cyclic garbage, a fact also used by subsequent work (see Section 2.4.2). Three interesting, previously established properties follow:

- decrements to non-zero reference counts are empirically known to be uncommon (which is why they are used to reduce the set of candidates in trial deletion);
- the condition is trivially identified by the reference counter during batch processing of decrements; and
- the condition is robust to coalescing of reference counts (exploited by Blackburn and McKinley [2003] and discussed by Paz et al. [2007]).

Cycle tracing therefore reduces its set of fix-up candidates to just those that experience decrements to non-zero reference counts, referred to as *purple* objects by Bacon and Rajan [2001]. Note that the correctness of this optimization to cycle tracing depends on purple object identification *during* heap tracing only, whereas trial deletion requires that this set be continually maintained. The results section shows that the overhead of continually maintaining the purple sets is measurable, so performing the operation only on demand may be sensible. However, the cycle tracing sweep optimization described below does depend on the purple set being continually maintained between invocations of the cycle collector.

4.1.4 Marking Optimization

Our second optimization is to reduce the scope of the mark phase by avoiding objects which are statically identified as being inherently acyclic. We use the simple method of determining acyclic classes in Java proposed by Bacon and Rajan [2001]. A class is said to be acyclic if it contains no pointer fields, or if it can point only to acyclic classes. Such objects are referred to as *green* objects, and are identified by a dedicated bit in each object's header. Bacon and Rajan use this bit to curtail the scope of each trial deletion trace to avoid tracing green objects. We trivially modify Step 2 in the base algorithm above to consider an object to be marked if the object is either marked *or* green. When using this optimization, green objects are not swept, but instead their collection is left entirely to the reference counter. The effectiveness of this optimization depends on the proportion of green objects in the heap. As shown later in Section 4.2.3, for many benchmarks a significant proportion of allocated objects are green.

4.1.5 Sweeping Optimization

Our third optimization is to limit the scope of sweeping to sweep only potentially cyclic objects and their children. We do this by using the same definition of potentially cyclic garbage as used in Section 4.1.3: objects subject to decrement to non-zero reference counts, which we refer to as purple. Rather than sweep the entire heap for unmarked objects, we note that the collector need only collect cyclic garbage, and therefore target our sweep at the potentially cyclic garbage identified by the purple set. This optimization is complete since all acyclic garbage will be collected by the reference counting collector.

4.1.6 Interaction With The Reference Counter

Cycle tracing places just three requirements on the underlying reference counting implementation. First, a consistent root set needs to be established. This is easily achieved by piggy-backing on an invocation of the deferred reference counter (which must establish a root set for its own purposes). The computation of this root set can be performed in either a stop-the-world or on-the-fly manner [Levanoni and Petrank, 2001]. Second, the fix-up set must be added to the gray queue (see Step 2(b) in Section 4.1.1) at a point where the set is known to be complete. This is trivial when the reference counter operates in a stop-the-world manner, which is the case in our system even though cycle collection work is concurrent. If all mutators are suspended, then it is sufficient to first process all increments and then all decrements before determining the fix-up set, with termination guaranteed if this fix-up set is empty. The third requirement is that the reference counter not free any objects known to the cycle tracing mechanism (which would cause cycle tracing to dereference a dangling pointer). Cycle tracing maintains a work queue (containing gray objects) and a fix-up queue (containing purple objects). The reference counter must simply mark any purple or gray objects for removal rather than directly freeing them; the

cycle tracer takes responsibility for freeing them as they are removed from the buffers and identified as free.

Both trial deletion and cycle tracing place similar requirements upon the underlying reference counter (including the establishment of the purple set), so it would also be possible to construct a system where the collectors are interchanged dynamically. While some variations of cycle tracing do not maintain a purple set, it is possible to establish an appropriate purple set during a complete cycle collection, making it possible to switch to these variations (or trial deletion).

4.1.7 Invocation Heuristics

A detailed analysis of heuristics for invoking cycle collectors is outside the scope of this work. A trivial policy is to invoke the cycle collector whenever the underlying reference counter is unable to free memory according to a specified *space usage threshold*. In collectors that rely on a purple set—such as trial deletion and some variants of cycle tracing—the size of the purple set might also constitute a suitable cycle detection trigger.

4.2 Evaluation

This section first briefly describes the platform within which all collectors have been implemented and evaluated. We then present the characteristics of the machines on which we conducted our experiments as well as some features of the benchmarks used.

We evaluated three cycle collectors: cycle tracing, simple backup tracing, and trial deletion, exploring performance in three ways:

1. performing a limit study of raw cycle collection throughput, where the cycle collector is forced to run at set intervals in a non-concurrent setting;

2. examining concurrency, measuring the efficiency of the cycle tracing concurrency optimization; and

3. comparing overall performance in a more natural setting, where the collectors are invoked only when deemed necessary by a cycle collection heuristic.

4.2.1 Implementation Details

We used MMTk in Jikes RVM version 2.3.4+CVS, with patches to support *replay compilation* [Blackburn et al., 2006]. MMTk is a flexible high performance memory management toolkit used by Jikes RVM [Blackburn et al., 2004a]. Jikes RVM is a high-performance virtual machine written in Java with an aggressive optimizing compiler [Alpern et al., 1999, 2000]. We use configurations that pre-compile as much as possible—including key libraries and the optimizing compiler—and turn

Table 4.1: Benchmark statistics showing total allocation, minimum heap size, percentage allocated green (acyclic), and percentage collected due to cyclic garbage.

Benchmark	Allocated (MB)	Min. Heap (MB)	Green	Cyclic
compress	116	14	91%	93%
db	90	16	11%	1%
jack	271	8	72%	2%
javac	241	20	47%	23%
jess	300	9	8%	2%
mpegaudio	5	8	6%	45%
mtrt	173	16	21%	6%
raytrace	163	12	20%	4%
pjbb2000	315	36	47%	14%
antlr	301	13	85%	13%
bloat	684	22	43%	12%
fop	66	24	69%	28%
hsqldb	592	21	65%	14%
jython	462	13	0.6%	4%
pmd	322	20	17%	24%
ps	572	9	46%	2%
xalan	77	99	89%	58%
Arithmetic Mean			**43%**	**20%**

off assertion checking (this is the *Fast* build-time configuration). The adaptive compiler uses sampling to select methods to optimize, leading to high performance but a lack of determinism. Since our goal is to focus on application and garbage collection interactions, we use the *replay* approach to deterministically mimic adaptive compilation.

By using the MMTk framework we are able to perform an apples-to-apples comparison of the collectors, with all base mechanisms shared by the different collector implementations. MMTk includes all space consumed by meta-data as part of overall memory consumption, including all work queues, buffers, and free list meta-data.

4.2.2 Experimental Platform

Experiments were conducted on a 2.2GHz AMD 64 3500+ running Linux 2.6.10. The data and instruction L1 caches are 64KB 2-way set associative. It has a unified, exclusive 512KB 16-way set associative L2 cache, and is configured with 2GB of dual channel 400 DDR RAM configured as 2 $\times$ 1GB DIMMs on an MSI nForce3 Ultra motherboard with an 800MHz front side bus.

4.2.3 Benchmarks

Table 4.1 shows the key characteristics of each of the 17 benchmarks used. These experiments were conducted with beta version `beta050224` of the subsequently re-

leased DaCapo benchmarks [Blackburn et al., 2006], a suite of non-trivial real-world open source Java applications. We also use the SPECjvm98 [SPEC, 1999] suite and pjbb2000, a variant of SPECjbb2000 [SPEC, 2001] that executes a fixed number of transactions to perform comparisons under a fixed garbage collection load. In Table 4.1, the *Allocated* column shows the total volume of allocation. The *Min. Heap* column shows the minimum heap size in which each benchmark runs successfully using the default configuration of MMTk (a generational mark-sweep collector). The *Green* column shows the fraction of allocated objects (in bytes) that were statically determined to be green (acyclic). The *Cyclic* column shows the fraction of data (in bytes) that a reference counter alone is unable to collect due to cyclic garbage. Note that this includes not only objects directly participating in garbage cycles, but also any objects kept alive by references from garbage cycles.

4.2.4 Throughput Limit Study

We begin by describing a limit study to analyze the fundamental efficiency of three cycle collectors: the base backup tracing algorithm, trial deletion, and cycle tracing. All three cycle collectors are correct and complete, and are able to collect any cyclic garbage present in a given heap. Abstractly, this approach exposes each collector to a large number of cycle collection opportunities and measures the efficiency with which they perform the collection task. Concretely, this is achieved by forcing the cycle collectors to collect a reference-counted heap after a fixed volume of allocation. In each case the cycle collector is invoked in a stop-the-world manner, thereby ignoring concurrency related concerns. This approach is simply an analytical tool, not a practical way to collect cycles. We examine the overall cost of the collectors in a more natural setting in the following sections.

Given that all collectors execute in a stop-the-world setting, this analysis is limited to examining the effectiveness of the mark optimization and sweep optimization as described in Sections 4.1.4 and 4.1.5. The effectiveness of the cycle tracing concurrency optimization is addressed in Section 4.2.5.

Figure 4.3 shows the overall throughput of the cycle tracing and trial deletion algorithms (normalized against backup tracing) for cycle collection invocation frequencies ranging from 128KB to 128MB. We show the average time for each cycle collection, taking the geometric mean of this value for all 17 benchmarks. These results show that cycle tracing outperforms the base algorithm by around 20%, while trial deletion performs around 70% worse than the base algorithm. This appears stable across the range of invocation frequencies examined, and while it is possible that this will not be the case for all invocation frequencies, note that the upper limit (128MB) is in the order of the total volume of allocation for many benchmarks (see Table 4.1).

Tables 4.2 and 4.3 (on page 52) show a breakdown of results for each of the 17 benchmarks when using an 8MB invocation frequency. Table 4.2 gives absolute figures for the base backup tracing algorithm, showing: cycle collection time (in milliseconds), nodes visited (in millions), and average cost per node visit (in nanosec-

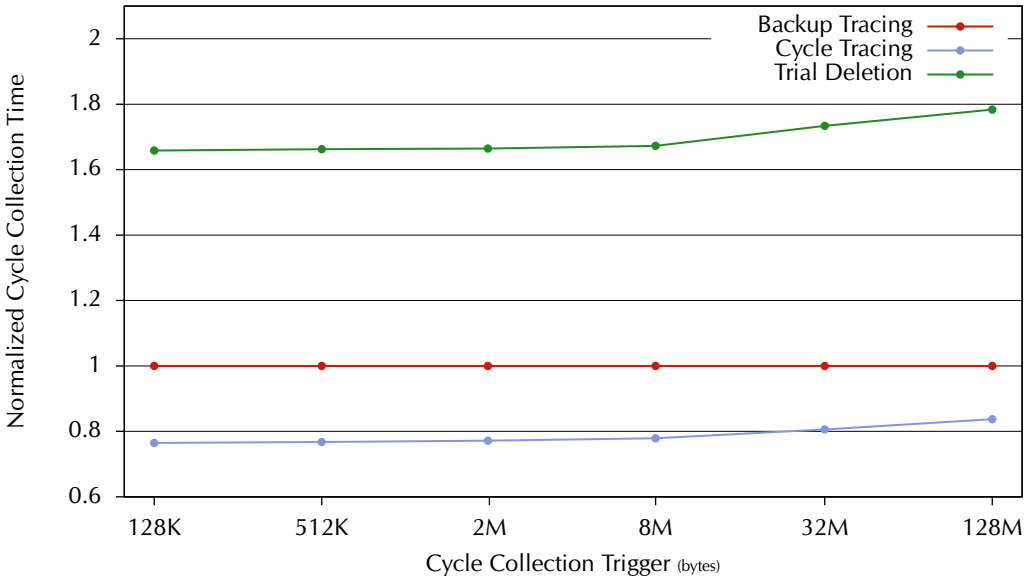


Figure 4.3: Throughput limit study with varying invocation frequency.

onds). Table 4.3 then shows the corresponding figures for cycle tracing and trial deletion, normalized against the results obtained for backup tracing. For all benchmarks, trial deletion is substantially and consistently slower than backup tracing, with the best result being a 41% slowdown in pjbb2000. In contrast, cycle tracing outperforms backup tracing in 16 of the 17 benchmarks, with only fop showing a 2% slowdown. Other benchmarks show consistent improvements of between 10% and 30%.

The *Visits* column for cycle tracing indicates that on average cycle tracing visits about 12% fewer nodes than backup tracing, demonstrating the efficacy of the mark optimization from Section 4.1.4. This reduction is not as significant as might be expected given the high proportion of green objects shown in Table 4.1. Two factors could account for this discrepancy. First, it is still necessary to visit the *green fringe*, that is the set of green objects that are directly referenced from non-green objects (it might be possible to further optimize the marking process by not considering object fields that always point to green objects). Second, the numbers in Table 4.1 reflect the total volume of *allocated* objects, whereas the counts given during execution are determined by the proportion of green objects in the heap at an instant in time.

The *Visit Cost* column for cycle tracing shows that the average cost of a node visit is also lower in cycle tracing than backup tracing. This is due to two factors. First, as mentioned above, the mark optimization means that visits to green objects are very fast, since no marking or scanning work is required. Second, because this figure is calculated by dividing the total cycle collection time by the number of nodes visited, and the cycle tracing sweep optimization reduces overall cycle collection time, the average visit cost is reduced.

Table 4.2: Throughput limit study showing average costs per cycle collection for backup tracing (invoked after every 8MB of allocation).

Benchmark	Backup Tracing		
	Time (ms)	Visits ($\times 10^6$)	Visit Cost (ns)
jess	89.98	11.62	7.74
raytrace	91.25	11.95	7.64
db	92.38	12.12	7.62
javac	118.33	14.59	8.11
mpegaudio	66.74	8.94	7.46
mtrt	99.75	12.90	7.73
pjbb2000	126.29	13.83	9.13
antlr	95.78	11.84	8.09
bloat	116.56	14.12	8.25
fop	108.97	12.94	8.42
hsqldb	104.29	13.01	8.02
jython	105.62	13.24	7.98
xalan	108.82	13.23	8.23
Arithmetic Mean	**101.9**	**12.64**	**8.03**

Table 4.3: Throughput limit study showing average costs per cycle collection for cycle tracing and trial deletion (invoked after every 8MB of allocation) normalized to backup tracing.

Benchmark	Cycle Tracing			Trial Deletion		
	Normalized to backup tracing			Normalized to backup tracing		
	Time	Visits	Visit Cost	Time	Visits	Visit Cost
jess	0.81	0.88	0.93	1.76	1.17	1.51
raytrace	0.78	0.87	0.90	1.64	1.28	1.28
db	0.73	0.85	0.86	1.52	1.15	1.32
javac	0.94	0.92	1.02	1.94	1.99	0.98
mpegaudio	0.77	0.90	0.86	1.65	1.23	1.34
mtrt	0.80	0.87	0.92	1.65	1.32	1.25
pjbb2000	0.71	0.84	0.84	1.41	1.55	0.91
antlr	0.91	0.90	1.01	1.86	1.40	1.33
bloat	0.81	0.91	0.89	1.65	1.55	1.07
fop	1.02	0.92	1.10	1.88	1.93	0.97
hsqldb	0.83	0.81	1.02	1.70	1.44	1.18
jython	0.81	0.90	0.90	1.70	1.30	1.30
xalan	0.86	0.90	0.95	1.73	1.49	1.16
Geometric Mean	**0.83**	**0.88**	**0.93**	**1.69**	**1.43**	**1.19**

4.2.5 Concurrency

All of the experiments described in the previous section were performed in a stop-the-world setting, and therefore precluded any analysis of the concurrency optimization from Section 4.1.3. Recall that the concurrency optimization allows a reduction in the size of the fix-up set—the set used by the snapshot-at-the-beginning collector to account for any collector–mutator races.

Initially, we expected the concurrency optimization to reduce the amount of work required, resulting in a direct improvement to overall efficiency. In practice, measurements in a uniprocessor time-slicing context showed that the optimization had no measurable impact.

Although Section 4.1.3 showed that decrements to objects whose reference count falls to zero are not *required* to ensure correctness, the structure of our collector means that the cost of performing fix-up operations on these objects is insignificant. The reference count of the object is stored in the same word as the cycle tracing mark state, and so the cost of checking the reference count is the same as directly checking the mark state.

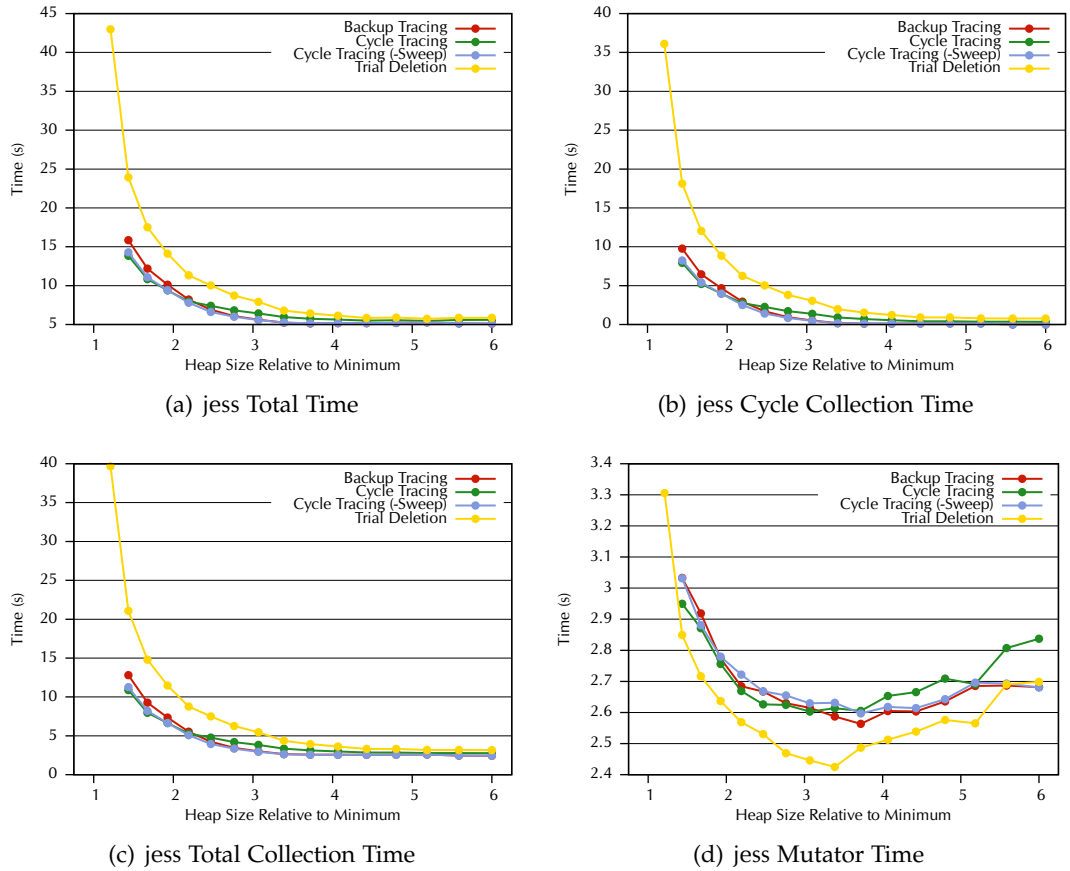
It is also worth noting that the amount of fix-up work required in any case was minimal, although one would expect the amount of work to increase in a truly concurrent setting (as opposed to time-slicing).

4.2.6 Overall Performance

This section evaluates the performance of the collectors. All experiments use a simple heuristic where cycle collection was triggered when the reference counter was unable to reclaim sufficient space in a fixed sized heap. All cycle collectors are invoked in a stop-the-world manner. We did not have a concurrent implementation of trial deletion, and we found in experiments with backup tracing and cycle tracing that the choice of heuristics dominated results, rather than the algorithm. A stop-the-world setting allows us to more fairly match the heuristics used across the different cycle collection approaches.

Figures 4.4–4.6 show more detailed performance results for three representative benchmarks, giving total time, overall garbage collection time (inclusive of cycle collection), cycle collection time, and mutator time as a function of heap size. Included are measurements for backup tracing, cycle tracing with mark and sweep optimizations, cycle tracing with just the mark optimization, and trial deletion. All of the graphs plot time in seconds. For javac, cycle collection costs become noticeable at heap sizes less than 2.5× the minimum, while for jess and bloat cycle collection costs are noticeable for heap sizes less than 4× the minimum (see Figure 4.4(b), Figure 4.5(b), and Figure 4.6(b)).

The most surprising result in Figures 4.4–4.6 is that the cycle tracing with mark and sweep optimizations performs worse than cycle tracing with just the mark optimization in all but the tightest heap sizes. The sweep optimization uses the purple set—containing roots of potentially cyclic garbage—to restrict the sweep to just those objects, avoiding a potentially expensive sweep of the entire heap. Closer analysis



(a) jess Total Time

(b) jess Cycle Collection Time

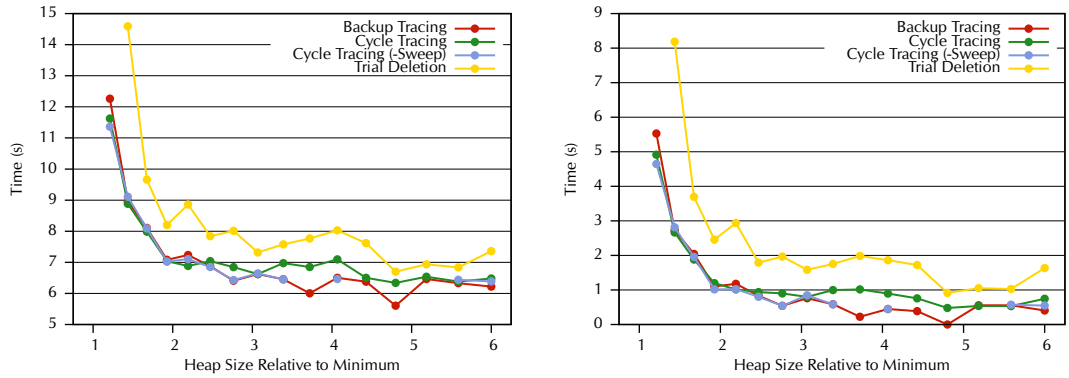
(c) jess Total Collection Time

(d) jess Mutator Time

Figure 4.4: Total, mutator, garbage collection, and cycle collection performance for jess.

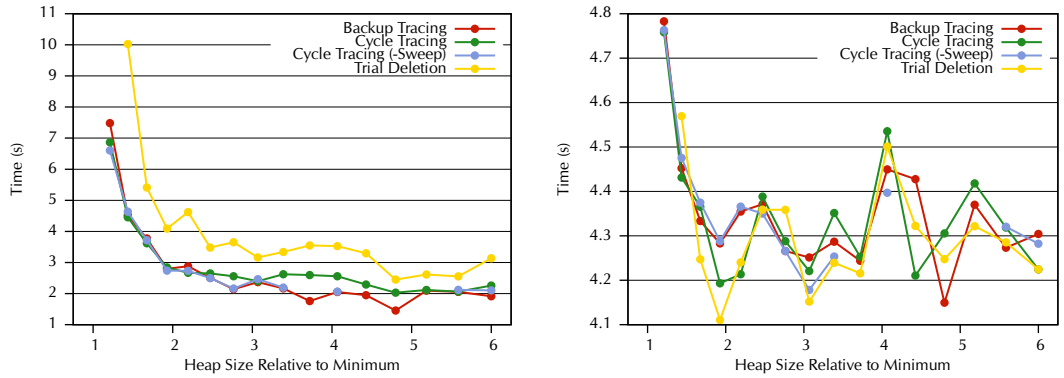
reveals that any advantage in a more targeted sweep is lost to purple set maintenance in large heap sizes. The correctness of the sweep optimization requires that the purple set contain *all* purple objects identified since the last cycle collection.

Purple set maintenance costs both space and time. Because we count all metadata (including the purple set) as heap memory, a large purple set can lead to heap pressure and consequently increased collector load, evident in Figure 4.4(b) and Figure 4.6(b). Trial deletion and cycle tracing (with mark and sweep optimizations) continue to perform measurable cycle collection work in large heaps, while the others perform none. The time overhead is due to the need to filter the purple set periodically to remove objects which have been collected by the reference counter. As the cycle collections become less frequent, the size of the purple set accumulates over a longer time, becomes larger, and requires more filtering, explaining why at the tightest heaps the sweep optimization is not harmful. The difference in performance across heap sizes suggests a more intelligent approach. Because the sweep optimization makes sense only when the purple set is small, a cap could be placed on the purple set size. Once the cap is exceeded, the purple set can be discarded (unmaintained until the next cycle collection phase), where the cap can then be reinstated and



(a) javac Total Time

(b) javac Cycle Collection Time



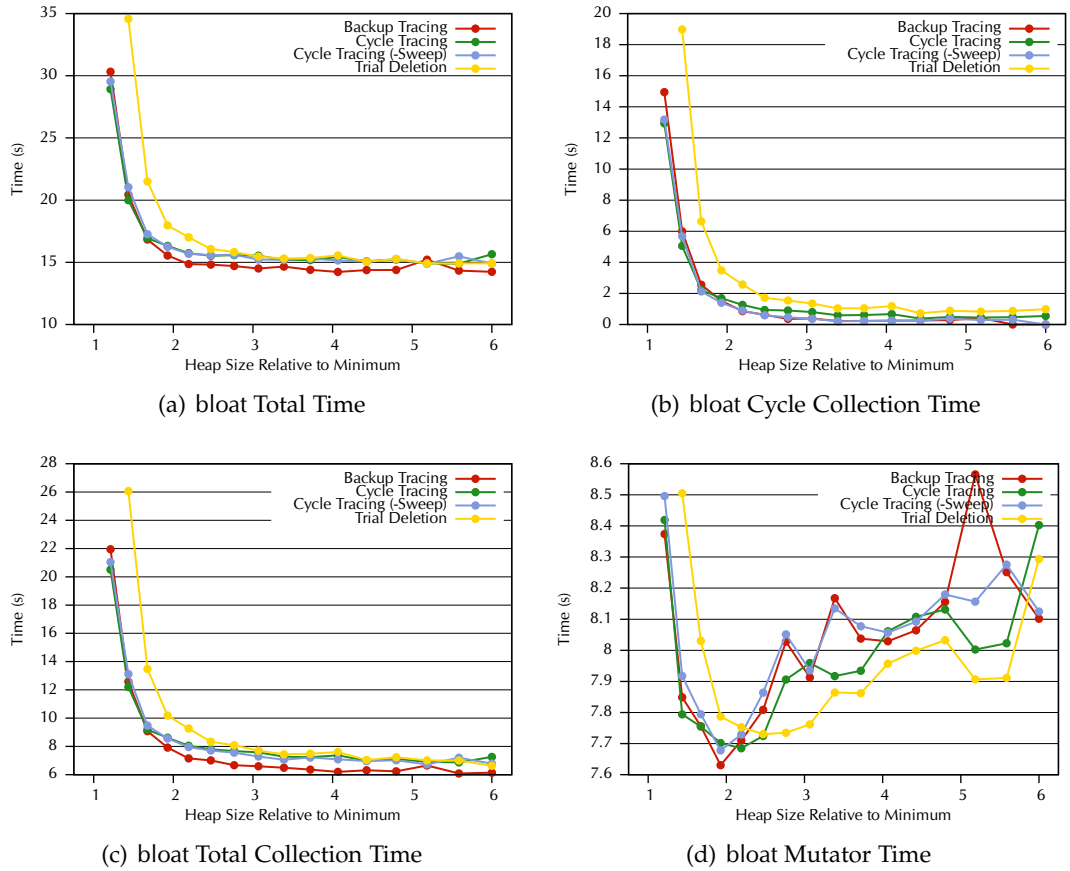
(c) javac Total Collection Time

(d) javac Mutator Time

Figure 4.5: Total, mutator, garbage collection, and cycle collection performance for javac.

the process started again. Such a hybrid would thus dynamically choose whether to use the sweep optimization. The overhead of dealing with the purple set could also be addressed by using a purple object bitmap. This would come at a constant space overhead, but would avoid the need for filtering. A combination of modest buffers and a bitmap updated by a single thread would avoid the need for atomic bitmap updates. Evaluating these alternatives is left to future work.

There are two other notable conclusions to be drawn from Figures 4.4–4.6. The first is that in benchmarks such as jess which allocate large numbers of short lived objects, the overhead of setting the gray bit on newly allocated objects is measurable. This is clear in Figure 4.4(d), where trial deletion holds a clear mutator time advantage over the others. Initial experience showed that setting the green bit in newly allocated acyclic objects was expensive, but we addressed this by modifying the Jikes RVM optimizing compiler to ensure that the green state is compiled in as a constant in the allocation sequence. The final result is perhaps the most striking of all, and that is the need for good heuristics for triggering of cycle collection. The heuristic is obviously naive; jess, which produces very little cyclic garbage, spends as much as half of its total running time performing cycle collection in tight heaps.



(a) bloat Total Time

(b) bloat Cycle Collection Time

(c) bloat Total Collection Time

(d) bloat Mutator Time

Figure 4.6: Total, mutator, garbage collection, and cycle collection performance for bloat.

4.3 Summary

This chapter introduced the novel tracing based cycle collection algorithm—*cycle tracing*—and described several optimizations targeted at improving cycle collection performance. Not all of the described optimizations improved performance, but the results obtained indicate that cycle tracing provides a significant performance improvement over state-of-the-art backup tracing and trial deletion cycle collectors.

Chapter 5

Generational Metronome

Chapter 2 describes Metronome [Bacon et al., 2003b]—the first collector to provide proven bounds on time, space, and utilization. While suitable for hard real-time applications, in Metronome these guarantees come at a significant cost to overall throughput performance. This chapter describes *Generational Metronome*, a fully incremental, real-time, generational collector based on a three stage nursery configuration. This collector achieves real-time bounds comparable to a non-generational Metronome-style collector, while cutting memory consumption and total execution times for generational workloads by as much as 44% and 24% respectively.

Section 5.1 highlights the features of the collector, and describes the real-time generational algorithm, focusing on key implementation and algorithmic hurdles. Section 5.2 introduces an analytical model for the collector, and describes the collector and application parameters required to provide real-time guarantees. Section 5.3.1 then presents a thorough evaluation of the generational collector, showing both analytically and experimentally that the collector achieves real-time bounds comparable to a non-generational Metronome-style collector, while cutting memory consumption and total execution time.

This chapter is based on work published in the paper "Generational Real-Time Garbage Collection: A Three-Part Invention for Small Objects" [Frampton, Bacon, Cheng, and Grove, 2007].

5.1 Real-Time Generational Collection

The potential for reducing memory consumption and/or improving throughput by employing a generational technique is well understood. A few generational collectors with various levels of soft- or hard-real-time behavior have been described [Bacon et al., 2005; Doligez and Leroy, 1993; Doligez and Gonthier, 1994; Domani et al., 2000b,a], but they either collect the nursery synchronously, or prevent nursery collections during mature collections—making it necessary for the mature collector to handle the full allocation rate of the application. This leads either to long pauses (in the order of 50 ms) or a very limited nursery size. For example, if the target maximum pause time is 1 ms and the evacuation rate is 100 MB/s, a synchronous nursery can be no larger than 100 KB. At such small sizes the survival rate is often too high

to derive much benefit from generational collection.

This chapter presents a fully generational version of the original Metronome real-time garbage collector [Bacon et al., 2003b], in which both nursery and mature collections are performed incrementally, and in which the scheduling of the two types of collections is only loosely coupled. This allows nursery collection to occur at any time, including in the middle of a full-heap collection.

Our generational algorithm is more complex but yields one significant advantage: greater flexibility in sizing the nursery while still meeting the real-time requirements of the application. The algorithm allows the collector to achieve very short pause times (nominally 500 μs) and reliable real-time behavior, while using a nursery large enough to achieve low survival rates.

The key contributions of this work are:

- An algorithm for fully generational real-time garbage collection in which both the nursery and major collections are incremental and can be arbitrarily interleaved.
- A nursery configuration based on a three stage life cycle that allows one nursery to be collected while the application continues to allocate into a separate nursery.
- An analysis of the space bounds and mutator utilization of a generational collector in which the nursery size is elastic, including the derivation of the nursery size that maximizes utilization and minimizes memory consumption.
- Measurements of applications showing that our generational collector is able to achieve real-time behavior comparable to a non-generational Metronome system, while using significantly less memory and increasing throughput.

5.1.1 Key Challenges

To make generational real-time collection more widely applicable, we must:

1. Make nursery collection incremental or concurrent; and
2. Ensure that nursery collections can always make progress (e.g., nursery collection should not be affected by the mature collection state).

Achieving both of these design goals decouples worst case pause time from nursery size, allowing the nursery size to be chosen to yield the low survival rates critical for effective generational collection.

The remainder of this section outlines the key challenges in incremental nursery collection and how our system addresses them. This is not a complete description of the generational algorithm, but covers all of the key extensions necessary to build an incremental generational collector on top of the base Metronome system. Section 5.1.3 introduces the three-part nursery configuration, which enables the mutator

to continue allocating while a nursery collection is in progress. Section 5.1.4 describes the techniques used to collect the nursery, including the write barriers used to establish and preserve the nursery root set. Finally, Section 5.1.5 discusses how the nursery and mature collections interact when both are active.

5.1.2 Basic Structure

The fundamental goal of our approach is to allow mutators to continue executing—*and therefore allocating*—while a nursery collection is taking place. The heap in our generational system consists of a single mature area, and a sequence of nurseries that follow a *three stage* life cycle, as described in the following section. Unlike the synchronous nursery collector, Syncopation [Bacon et al., 2005], each nursery in our system is not fixed in size. Instead, each nursery continues to grow due to mutator allocation until it is both *desirable* and *possible* to begin a nursery collection cycle.

It is considered *desirable* to initiate a nursery collection once a certain amount of allocation—the *nursery trigger*—has occurred. The nursery trigger is a system parameter that can be varied to trade off survival rate with memory consumption (see Section 5.2).

However, it is not *possible* to begin a nursery collection if a previous nursery collection is still in progress, so a new collection will be deferred while the previous collection completes. The mutator continues to allocate during this period, making it possible for the nursery to exceed the size specified by the nursery trigger. This introduces some degree of nursery *elasticity*, allowing the system to smoothly absorb short-term spikes in the application allocation rate, without resorting to direct allocation into the mature area (which defeats any attempt to improve guaranteed real-time behavior through the nursery).

Nursery pages are allocated out of a single global pool of pages shared with the mature area. This facilitates the development of a simple model of the system. Allocation to nursery pages is performed using a simple bump pointer. Objects are promoted by copying them into cells allocated in the mature space using a segregated free list allocator.

As is typical for generational collectors, we use a write barrier to capture references created from the mature space to a nursery object. These references are stored in a remembered set. As with nurseries, there is a sequence of remembered sets, each associated with a single nursery. Figure 5.1 shows the section of the write barrier that is inlined into application code. This barrier differs from the non-generational Metronome barrier only through the additional check on line 6; both the forwarding of the target value on line 3, and the check against collector tracing state on line 5 are unchanged. During both nursery and mature collections `collector_tracing` is set to **`true`** to ensure that a correct snapshot-at-the-beginning collection is performed.

```
1  writeBarrier (Object source, Address slot, Object target) {
2    // Ensure forwarded
3    target = forward(target);
4
5    if (collector_tracing ||
6        (isMature(source) && isNursery(target)) {
7      slowPath(source, slot, target);
8    }
9  }
```

Figure 5.1: Generational Metronome write barrier pseudo-code.

5.1.3 Three Stage Nursery Life Cycle

As described in the previous section, our heap consists of a single mature space, **M**; and a sequence of nurseries, $\mathbf{N_0}$ $\mathbf{N_1}$ $\mathbf{N_2}$ ... $\mathbf{N_k}$, which each progress through a *three stage* life cycle:

1. ***allocate.*** Each nursery starts its life as the *allocate* nursery.
2. ***collect.*** When it is time to collect the *allocate* nursery, it becomes the *collect* nursery, and a new *allocate* nursery is created.
3. ***redirect.*** After a nursery collection is completed, the *collect* nursery is retained as the *redirect* nursery, because references could have been created to it during collection, and these references might not be discovered until the collection of the *next* nursery in the sequence.

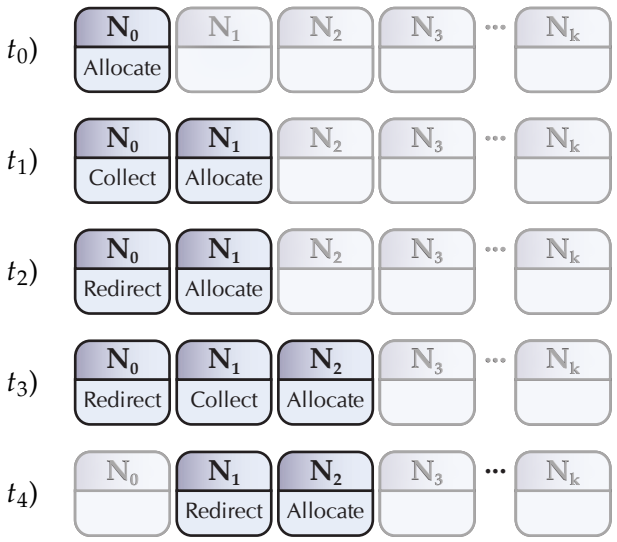


Figure 5.2: Initial sequence of nurseries progressing through the three stage life cycle.

At any point in time there will exist at most 3 nurseries, consisting of:

- a single *allocate* nursery;
- a single *collect* nursery, *whenever a nursery collection is in progress*; and
- a single *redirect* nursery, *at any point after the first nursery collection*.

The initial sequence of nurseries passing through these stages is illustrated by Figure 5.2. Each nursery, $\mathbf{N_k}$, has associated with it a remembered set, $\mathbb{R}_\mathbf{k}$, which captures all mature to nursery references created *while* $\mathbf{N_k}$ *is the allocate nursery*. As described in the following sections, this may include references to the *collect* nursery.

5.1.4 Incremental Nursery Collection

This section describes the process of performing a single incremental nursery collection. First, Section 5.1.4.1 describes references the mutator can create *outside* of a nursery collection. Next, Section 5.1.4.2 describes the process of *initiating* a nursery collection. Then, Section 5.1.4.3 describes mutator and collector activities *during* a nursery collection. Last, Section 5.1.4.4 describes the state of the system at the conclusion of a nursery collection.

5.1.4.1 Outside Nursery Collection

Figure 5.3 shows the references that can be created during mutator intervals *outside* of a nursery collection. These are simply references between and within $\mathbf{M}$ and $\mathbf{N_k}$ (the *allocate* nursery), with all references from the mature space to the nursery captured in remembered set $\mathbb{R}_\mathbf{k}$.

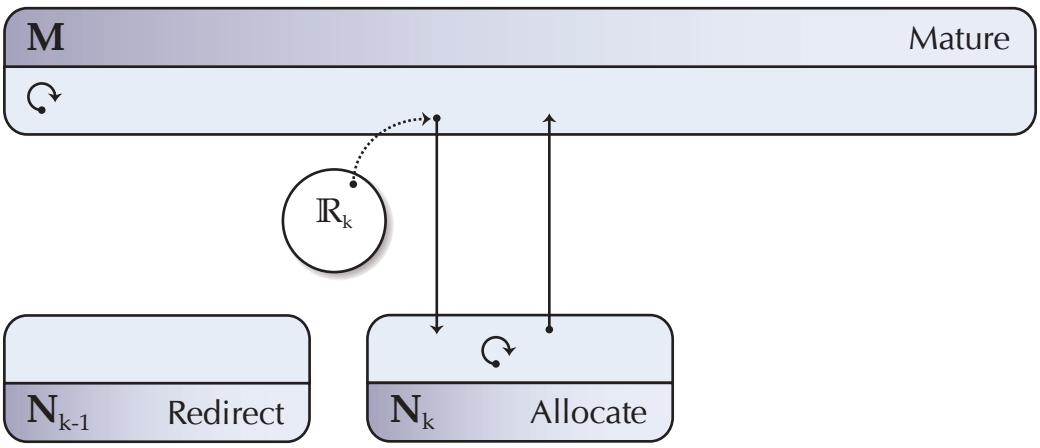


Figure 5.3: References created outside of a nursery collection.

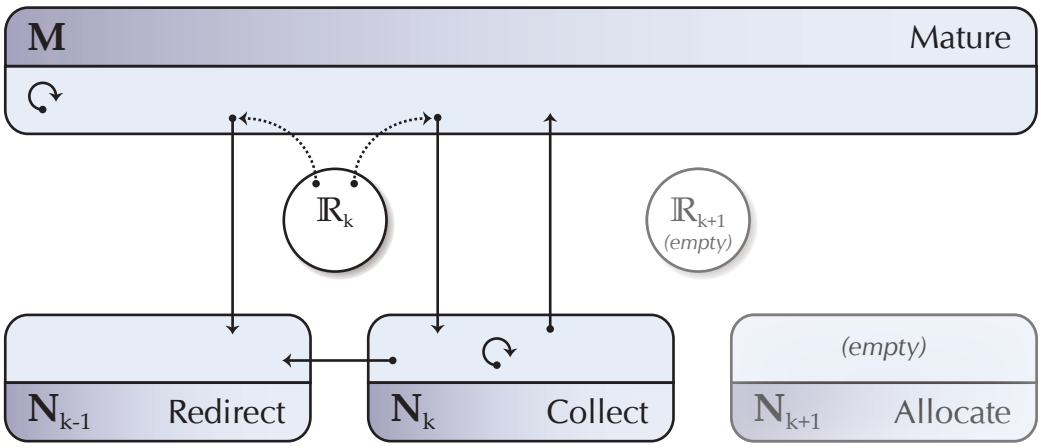


Figure 5.4: State at the start of the collection of nursery $\mathbf{N_k}$.

5.1.4.2 Start of Nursery Collection

Figure 5.4 shows the state of the system at the start of the collection of nursery $\mathbf{N_k}$. For the first collection, the *redirect* nursery $\mathbf{N_{k-1}}$ will not exist (and can be treated as empty) and the remembered set associated with the *collect* nursery, $\mathbb{R}_\mathbf{k}$, will be empty, because no objects have yet been promoted, and thus no references can exist from the mature space into a nursery. As Figure 5.4 shows, for subsequent collections, $\mathbb{R}_\mathbf{k}$ contains all of the references created from $\mathbf{M} \rightarrow \mathbf{N_k}$ (when $\mathbf{N_k}$ was the *allocate* nursery) and from $\mathbf{M} \rightarrow \mathbf{N_{k-1}}$ (when $\mathbf{N_{k-1}}$ was the *collect* nursery).

To initiate the collection of nursery $\mathbf{N_k}$, the following steps are performed *atomically*:[1]

- Close the remembered set $\mathbb{R}_\mathbf{k}$;
- Capture all root references to $\mathbf{N_k}$ in the root set, $\mathbb{Z}_\mathbf{k}$;
- Direct all mutators to begin allocating into $\mathbf{N_{k+1}}$;
- Direct all mutators to begin contributing remembered set entries into $\mathbb{Z}_\mathbf{k+1}$; and
- Activate the tracing write barrier to ensure the nursery is traced consistently.

Note that $\mathbb{R}_\mathbf{k} \cup \mathbb{Z}_\mathbf{k}$ provides a complete root set for the collection of nursery $\mathbf{N_k}$. *All* live objects in $\mathbf{N_k}$ are transitively reachable from $\mathbb{R}_\mathbf{k}$ and/or $\mathbb{Z}_\mathbf{k}$. By enabling a standard snapshot-at-the-beginning tracing barrier [Yuasa, 1990], it is possible to allow the mutator to execute while safely and completely collecting the nursery $\mathbf{N_k}$.

[1] To facilitate the exposition of the novel aspects of our collector, we describe the algorithm as if an atomic snapshot of the roots were taken. In practice, the root set can be captured incrementally using the approach of Azatchi et al. [2003].

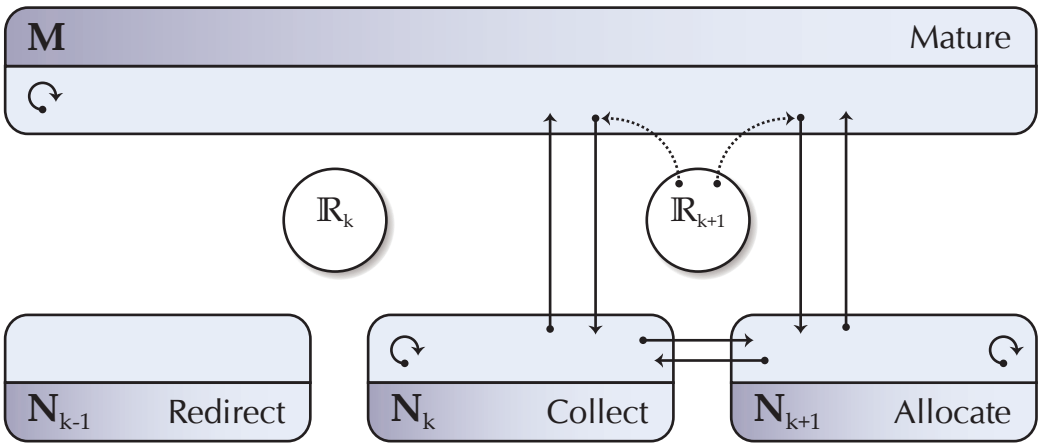


Figure 5.5: References that may be created during the collection of nursery $\mathbf{N_k}$.

5.1.4.3 During Nursery Collection

This section describes *collector* and *mutator* activity during an incremental nursery collection.

Collector activity. Collector threads perform a transitive closure over the *collect* nursery, starting with the root set $\mathbb{R}_\mathbf{k} \cup \mathbb{Z}_\mathbf{k}$, and maintaining a work list of gray objects. The work list contains objects that have been promoted, but have not yet had outgoing references scanned. During scanning, outgoing edges are treated according to the location of the target object:

- References to the mature space are ignored.
- References to the *redirect* nursery are redirected to promoted versions of the objects in the mature space.
- References to *marked* objects in the *collect* nursery are redirected to the promoted copy in the mature space.
- References to *unmarked* objects in the *collect* nursery result in the object being promoted to the mature space, with the original reference redirected to the promoted copy in the mature space.
- References to the *allocate* nursery, $\mathbf{N_{k+1}}$, are added to the active remembered set, $\mathbb{R}_{\mathbf{k+1}}$.

Mutator activity. Mutators are free to resume executing while the *collect* nursery is being collected. To ensure the collector traces a valid snapshot of the heap, the mutator is required to execute the snapshot write barrier if a nursery collection is in progress. References that can be *created* by the mutator are shown in Table 5.1 and

Table 5.1: Handling of reference mutations during nursery collections.

Source \ Target	$\mathbf{M}$	$\mathbf{N_{k-1}}$	$\mathbf{N_k}$	$\mathbf{N_{k+1}}$
$\mathbf{M}$	1,2	5	$\mathbb{R}_{\mathbf{k+1}}$ *	$\mathbb{R}_{\mathbf{k+1}}$
$\mathbf{N_{k-1}}$	2,3,4	1,3,4,5	3,4	3,4
$\mathbf{N_k}$	2,3	3,5	1,3	3 †
$\mathbf{N_{k+1}}$	2,3	3,5	3 ‡	1,3

Reason reference is ignored

1 Intra-area reference.
2 Reference *into* $\mathbf{M}$.
3 Reference *from* $\mathbf{N}_*$.
4 Nursery $\mathbf{N_{k-1}}$ is *immutable*.
5 Reachable objects in $\mathbf{N_{k-1}}$ have been promoted.

Notes

* References from $\mathbf{M}$ to $\mathbf{N_k}$ must also have been obtained from the snapshot, and are therefore not required to correctly collect $\mathbf{N_k}$.

† References from $\mathbf{N_k}$ to $\mathbf{N_{k+1}}$ will be added to the remembered set by the collector as the source objects are promoted.

‡ References from $\mathbf{N_{k+1}}$ to $\mathbf{N_k}$ are not needed to correctly collect $\mathbf{N_k}$ (they must have been obtained from the snapshot given by $\mathbb{R}_\mathbf{k} \cup \mathbb{Z}_\mathbf{k}$) and so are equivalent to references from $\mathbf{N_{k+1}}$ to $\mathbf{M}$.

illustrated in Figure 5.5. Table 5.1 also indicates which references are added into the remembered set, and indicates why other references are not required for collection.

As Table 5.1 shows, references created into any nursery ($\mathbf{N_k}$ or $\mathbf{N_{k+1}}$) are added to the generational remembered set $\mathbb{R}_{\mathbf{k+1}}$. References can not be created to $\mathbf{N_{k-1}}$ because all references are forwarded as they are written to ensure tracing progress, and all objects in $\mathbf{N_{k-1}}$ that have references to them must have been promoted. References from $\mathbf{M}$ to $\mathbf{N_{k+1}}$ stored in $\mathbb{R}_{\mathbf{k+1}}$ are equivalent to the usual generational remembered set. References from $\mathbf{M}$ to $\mathbf{N_k}$ are not required for correct collection of $\mathbf{N_k}$, because $\mathbb{R}_\mathbf{k} \cup \mathbb{Z}_\mathbf{k}$ provides a complete root set for collecting $\mathbf{N_k}$, but such references will need to be updated once the target objects are promoted from $\mathbf{N_k}$. Updating these references is one reason why the *redirect* nursery is required.

5.1.4.4 End of Nursery Collection

Once the nursery collection is completed, the remembered set $\mathbb{R}_\mathbf{k}$ is fully processed and empty, and all objects in the *collect* nursery $\mathbf{N_k}$ that were transitively reachable from the mature space and/or $\mathbb{Z}_\mathbf{k}$ have been promoted. In addition, there are now no references to objects in the *redirect* nursery $\mathbf{N_{k-1}}$. These references were previously in the remembered set or in the *collect* nursery, $\mathbf{N_k}$. Because the remembered set $\mathbb{R}_\mathbf{k}$ in now empty, and $\mathbf{N_k}$ has been completely traced, no remaining references to the *redirect* nursery $\mathbf{N_{k-1}}$ remain. This allows the *redirect* nursery $\mathbf{N_{k-1}}$ to be reclaimed, and the *collect* nursery $\mathbf{N_k}$ to change role and take its place as the new *redirect* nurs-

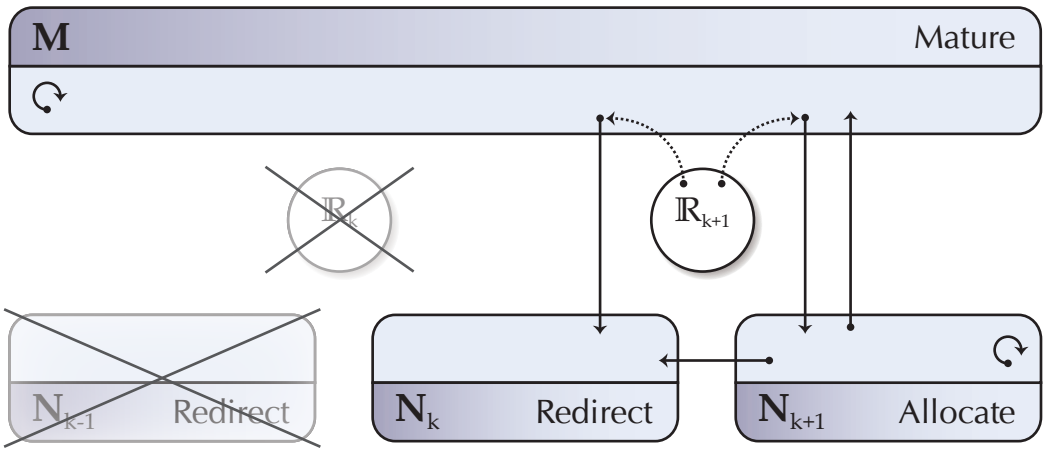


Figure 5.6: References that may exist *after* collection of nursery $\mathbf{N_k}$ is complete.

ery. The state of the system at this point is shown in Figure 5.6. From this point note that the only required information in the *redirect* nursery is forwarding pointer information for promoted objects. All live objects have been identified and promoted to the mature space.

Having described the complete process of performing a nursery collection in the previous sections, we next discuss some of the more interesting issues that arise when combining mature and nursery collections into a single system.

5.1.5 Mature–Nursery Collection Interactions

The previous section described the operation of the nursery collection, but additional complexities arise when integrating the mature and nursery collections. In order to provide the necessary real-time guarantees, the ability for a nursery collection to proceed must not be impeded by the mature collection process. In our system, the nursery collection is allowed to *preempt* a mature collection at any time. This requires the mature collection to maintain the system in a state where nursery collection is possible at the end of *every* mature increment. There is no inverse requirement; a correct parametrization of the system guarantees that the nursery will leave sufficient collection time for mature collection work.

To support independent mature and nursery collection, each object is allocated with two independent mark-bit fields. One field holds the object's nursery mark state, while the other holds the object's mature mark state. Each thread remembers the current values for each of the mark-bit fields to apply to newly allocated objects. The current values for the mark-bit fields are then updated as the thread is scanned at the beginning of a nursery or mature collection.

5.1.5.1 Mature Collector References to the Nursery

During a mature collection the mature collector potentially holds references to the nursery in two locations: 1) the root set used to establish the mature collection snapshot, and 2) the buffer filled with values captured by the snapshot write barrier. When a nursery collection preempts a mature collection, it is important to ensure that these references are processed to avoid creating dangling references.

To allow correct tracing by the mature collector, the nursery collector uses all values held by the mature collector as additional roots, to ensure that any portion of the nursery that is part of the mature snapshot is kept alive. In order to allow this to occur without forcing the nursery collector to process the entire mature collector's buffer, the mature space is required to maintain nursery references separately. This splitting is performed both as roots are calculated, and as the snapshot write barrier discovers unmarked objects.

5.1.5.2 Mark State of Promoted Objects

The nursery must promote objects into the mature space in a consistent state. If, for example, the nursery were to promote objects into the mature space as unmarked after mature tracing is complete, the mature space might sweep up these live objects. Similarly, if the nursery were to promote objects as marked during tracing, references from these objects to the mature space might be missed by the mature collector, causing live mature objects to be collected.

Maintaining correct mark state at promotion is the motivation for associating two mark-bit fields with each object. All objects, including those in the nursery, have a valid mature mark-bit field. In line with the allocate-black property of the mature snapshot collector, the mature mark-bit field records objects allocated after the mature collection has started as marked. When the nursery promotes objects, the value of this mark-bit field is preserved, ensuring that the mature trace and sweep progress correctly. This makes it possible for the nursery to promote an object that will *never* be marked by the current mature collection, but because nursery collections preempt mature collections, the mature collection can never free such an object while the nursery is still processing it.

5.1.5.3 Sweeping Objects Stored in Remembered Sets

In general, it is possible for objects referred to by the remembered sets to become garbage. While a stop-the-world generational collector can simply discard all remembered set entries during a mature collection, this is not possible with incremental generational collection. Unless these remembered set entries are handled, the nursery collector is potentially exposed to invalid data. In order to resolve this issue in our system, we require the mature collector to sweep the remembered set of all references to garbage objects *before* the space associated with the objects is freed. This allows the nursery collection to preempt the mature collection at any time, because

either the remembered set entry will have been removed, or the object will still be present in memory.

5.2 Analytical Model

To formalize the performance of our generational collector, this section introduces an extension of the model used by Metronome [Bacon et al., 2003a] to include generational collection. This model is the basis under which real-time performance guarantees can be provided.

Generational collection provides greater efficiency by focusing work on an area where there is a high proportion of dead objects. However, in the case where the nursery survival rate is high, then the additional cost of copying each live object out of the nursery will make the generational collector perform comparatively poorly.

5.2.1 Definitions

The garbage collector is characterized using the following parameters:

R_T is the tracing rate in the heap (bytes/second);

R_S is the sweeping rate in the heap (bytes/second); and

R_N is the collection rate in the nursery (bytes/second).

The application is characterized by the following parameters:

a is the maximum allocation rate in mutator time (bytes/second) within a time window required for a mature collection cycle (i.e., the maximum average number of bytes allocated for every second of processor time that the mutator is active);

m is the maximum live memory size of the mutator in bytes; and

$\eta(N)$ is a function that provides the maximum survival rate of a nursery of size *at least* N bytes (taking into account the generational barrier). By this definition $\eta(N)$ is monotonically decreasing in N (eliminating quantization effects caused by irregular survival patterns).

The real-time behavior of the system in characterized by the following parameters:

Δt is the task period in seconds; and

u is the minimum mutator utilization or MMU [Cheng and Blelloch, 2001] for the task period Δt.

5.2.2 Steady-State Assumption and Time Conversion

While the allocation rate and survival rate can vary considerably during execution, we start by considering the case when they are smooth. However, dynamic nursery size variation is central to our approach, so we model it dynamically. As in previous Metronome collectors, modeling relies on being able to relate total time, mutator time, and collector time. For a given total time interval Δt, the collector may consume up to $(1-u)\cdot\Delta t$ seconds for collection. We define the *garbage collection factor* γ as the ratio of mutator execution to collector work:

$$\gamma = \frac{u\cdot\Delta t}{(1-u)\cdot\Delta t} = \frac{u}{1-u} \tag{5.1}$$

Multiplying by γ converts collector time into mutator time, and dividing does the reverse. Since the relationship between u in the range $[0,1)$ and γ in the range $[0,\infty)$ is one-to-one, we also have:

$$u = \frac{\gamma}{1+\gamma} \tag{5.2}$$

From the above parameters, we can now derive the overall space consumption of the system. Fundamentally, for all real-time collectors, the space requirements depend on the amount of extra memory that is allocated during the time when incremental collection is being performed and the mutator is continuing to run. Thus:

- s is the space requirement of the application in our collector; and
- e is the extra space allocated by the mutator over the course of a full-heap collection.

We first review bounds for s and e for previous collectors, and then show how they relate to the generational metronome collector.

5.2.3 Bounds for Non-Generational Metronome Collectors

In the absence of generational collection, the extra space e_M required by Metronome during a collection in order to allow that collection to complete (as described in [Bacon et al., 2003b]) is:

$$e_M = a\gamma\cdot\left(\frac{m}{R_T}+\frac{s_M}{R_S}\right) \tag{5.3}$$

This corresponds to allocation at the maximum rate across a complete collection cycle, and is expressed in mutator time. In practice, the total space requirements are greater.

Consider the case of a collection cycle that starts with m live memory, and has e_M memory allocated during the collection (and also e_M memory freed to satisfy m as the maximum live size). In this case, memory usage at the completion of the

collection cycle is $m + e_M$. A subsequent collection cycle must *also* be able to allocate an additional e_M of memory, so the extra space required must be at least $2e_M$.

The $2e_M$ extra space requirement is typical of incremental tracing collectors, but Metronome's approach to defragmentation leads to further space overheads. Any objects being defragmented occupy *twice* the space, because both the original version and the copy are stored. Clearly there is an additional extra space requirement equal to the maximum number of bytes being defragmented. Metronome sets this maximum value to e_M, because this is the minimum amount Metronome can defragment while still ensuring that there is sufficient space for arbitrary allocation of e_M memory in the subsequent collection cycle. This leads to a total additional space requirement of $3e_M$. We then adjust this by the rate of internal fragmentation due to the design of the segregated free list yielding:

$$s_M = (m + 3e_M) \cdot (1 + \rho) \tag{5.4}$$

Although in the common case space usage will be less, this is the tightest bound that can be placed on maximum space requirements. Note that the bound for e_M shown above depends on s_M. This circular dependency makes it necessary to either: ignore the cost of sweeping given tracing costs dominate (as in the original Metronome paper); conservatively overestimate the cost of sweeping (as in Bacon et al. [2003a]); or solve Eq. (5.4) for s_M.

5.2.4 Bounds for Our Generational Collector

Given a fixed time budget for total collection activity, time spent collecting the nursery in a generational collection reduces the amount of time available for full-heap collection, reducing the rate of full-heap collection progress. This in turn means that the mutator is able to perform more allocation during a full-heap collection cycle. However, generational collection also serves to attenuate allocation into the mature area by the survival rate $\eta(N)$. This effect is shown in Figure 5.7, and expressed by the following equations, which introduce the *generational dilation factor* δ, and the corresponding additional space requirement e_G with generational collection:

$$\delta = 1 - \frac{a\eta(N)}{R_N} \cdot \gamma \tag{5.5}$$

$$e_G = \frac{a\eta(N)\gamma}{\delta} \cdot \left(\frac{m}{R_T} + \frac{s}{R_S} \right) \tag{5.6}$$

Since our generational collector is fully incremental, we can maintain real-time behavior without restricting nursery size, and therefore use a nursery size which is best suited to the survival rate of the application. However, this flexibility comes with additional complexities in determining what that size should be.

Since $\eta(N)$ is monotonically decreasing and low values are crucial to the success of generational collection, let us consider what happens as the nursery size varies.

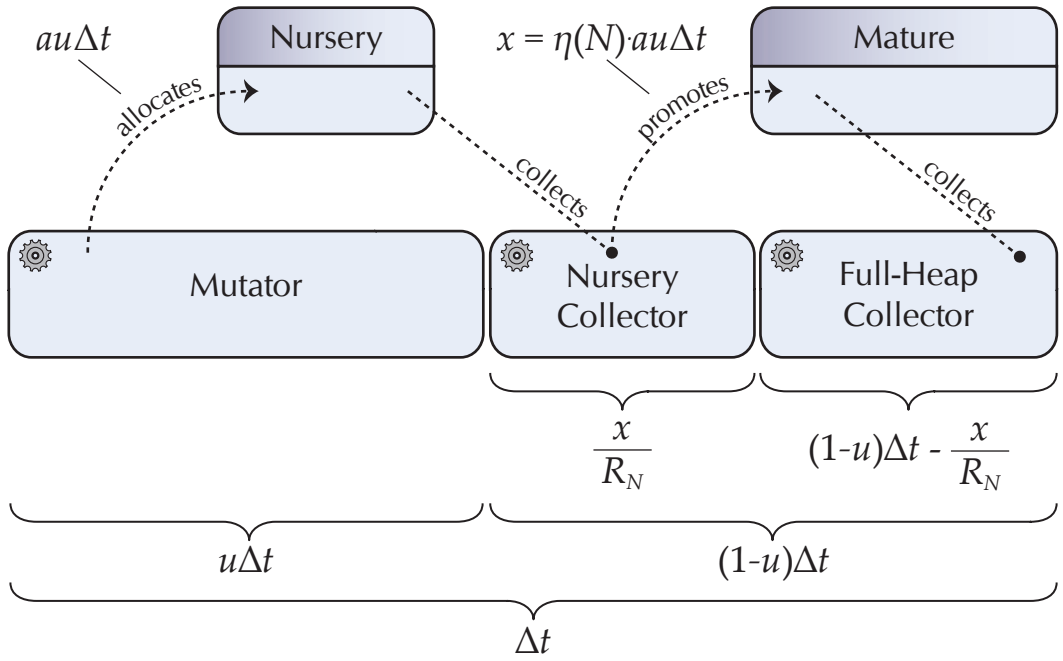


Figure 5.7: Time dilation due to generational collection causes additional allocation during a major heap collection, but attenuates all allocation by the survival rate $\eta(N)$.

For very small nursery sizes, the collector will spend more time performing nursery collections because the survival rate will be higher, forcing significant copying work. In practice, due to the elasticity of the nursery, when the nursery size is set very small, generational collection will reach a steady-state behavior where it is continuously collecting the nursery, using all available collection time for nursery collections:

$$\frac{N \cdot \eta(N)}{R_N} = \gamma \frac{N}{a} \tag{5.7}$$

In other words, N grows until it reaches a minimum tenable size N_{min}:

$$\eta(N_{min}) = \frac{\gamma \cdot R_N}{a} \tag{5.8}$$

Note that this requires that the nursery tracing rate R_N be at least $(a\eta(N))/\gamma$.

When the nursery size is set above this threshold, major collections are given an opportunity to complete, and memory is bounded. If the nursery is set arbitrarily large, overall memory consumption will increase because the nursery dominates the mature space in size. Between these two extremes is a nursery size which minimizes the overall heap consumption. In order to compute this point—and to compare the generational system against the non-generational system—we need to compute the space bounds of the system.

Because the generational version has the additional space cost of the three-part nursery, the total space requirement for a given application is:

$$s_G = (m + 3e_G) \cdot (1 + \rho) + 3N \tag{5.9}$$

The $(1 + \rho)$ factor is not needed in the $3N$ term as there is no fragmentation in the nursery—all objects are allocated contiguously in memory using a bump pointer. As shown in Eq. (5.8), the nursery will naturally grow in size until it meets the N_{min} threshold. If the nursery is larger than this crossover point, the heuristic will not grow the nursery further. However, continuing to grow the nursery above this size will actually diminish overall heap consumption. Just above the crossover point, the term δ is infinitesimally small, making s_G arbitrarily large. Similarly, as N approaches infinity, s_G is arbitrarily large. Thus, if we hold utilization constant (because it is a target), there must exist, by continuity, a globally minimal overall heap size for some nursery size. Inverting the function to express utilization in terms of s_G gives the achievable utilization for a particular overall heap size.

Note that we are making a steady-state assumption about $\eta(N)$. Since we are collecting the nursery itself incrementally and therefore handle a wide range of nursery sizes, this is reasonable for a large class of real programs. However, there is also a class of programs that have a setup phase which precedes a steady-state phase. For such programs the steady-state assumption—if applied to the entire program—may result in overly large nurseries. We will study an example of such a program in Section 5.3.4. This effect is also present in non-generational real-time collectors, but

is exacerbated in generational collectors. For both types of systems, it is desirable to allow the application to explicitly delineate the setup and mission phases, and to either allow real-time bounds to be violated during the setup phase in favor of reduced memory consumption, or to perform a (potentially synchronous) memory compaction between the two phases.

5.2.5 Comparison with Syncopation

Generational collection in a Metronome-style collector was previously described using a technique called *Syncopation* [Bacon et al., 2005]. Syncopation uses stop-the-world collection of the nursery combined with *flood-gating*—direct allocation into the mature space—when allocation and survival rates are too high for synchronous collection to be performed without violating real-time bounds.

The nursery size N with Syncopation was severely restrained, because the synchronous nursery collection placed severe bounds on real-time behavior. For the small nursery sizes that were feasible for synchronous nursery collection, real-world programs almost always have spikes in the survival rate, pushing the survival range $\eta(N)$ to 1. Therefore it is generally necessary to use the largest possible nursery size such that:

$$\frac{N}{R_N} = (1-u)\Delta t \tag{5.10}$$

$$N = (1-u)\Delta t R_N \tag{5.11}$$

The time dilation and extra space calculations then become simpler, such that:

$$\delta' = 1 - \frac{N}{R_N} \cdot \gamma \tag{5.12}$$

$$e_S = \frac{a\gamma}{\delta'} \cdot \left(\frac{m}{R_T} + \frac{s}{R_S} \right) \tag{5.13}$$

with the space bound for synchronous nursery collection:

$$s_S = (m + 3e_S) \cdot (1+\rho) + (1-u)\Delta t R_N \tag{5.14}$$

Although there is no factor of 3 multiplier on the nursery as for our generational collector (Eq. (5.9)), the higher survival rates incurred by the much smaller nurseries mean that the space consumption in the mature space increases significantly.

5.3 Evaluation

We implemented the generational algorithm as a modification to the IBM WebSphere Real Time Java virtual machine [IBM, 2006],[2] which uses the non-generational Metronome-based algorithm described in Section 2.5.1. Both collectors support the complete Java semantics, including finalization and weak/soft/phantom references.

We did not experimentally compare our system with Syncopation [Bacon et al., 2005]; the nursery sizes required to achieve low survival rates on non-embedded applications—in the order of 1MB for SPECjvm98—would incur pauses in the Syncopation system of at least an order of magnitude beyond the worst-case latencies for the other systems.

All experiments were run on an IBM Intellistation A Pro workstation with dual AMD Opteron 250 processors running at 2.4 GHz with a 1 MB L2 data cache. Total system memory was 4 GB RAM.

The operating system was IBM's real-time version of Linux[3] based on Red Hat Enterprise Linux 4. This includes a number of modifications to reduce latency, in particular the `PREEMPT_RT` patch with modifications for multicore/multiprocessor systems.

We begin our evaluation by showing a performance comparison of the generational and non-generational systems across a range of benchmarks. We then demonstrate the effectiveness of the dynamic nursery size at coping with short bursts of allocation. Selecting a highly generational benchmark, jess, highlights the importance of the large nursery sizes made possible through incremental nursery collection. We then discuss the difficulties in evaluating real-time collectors by observing differences between start-up and steady-state behavior.

The focus of our investigation is comparing collector performance, so we use a modified *second run* methodology. This methodology involves invoking a benchmark twice within a single JVM invocation; the first *warmup* run performs compilation and optimization, while results are gathered from a second *measurement* run.

The just-in-time (JIT) compiler implementation in the system is not real-time, so it is necessary to disable it during the measurement run. Between the warmup and measurement runs the JIT is disabled by calling `java.lang.Compiler.disable()` and then pausing to allow the compilation queue to drain. IBM's real-time JVM also includes an ahead-of-time compiler which could be used to factor out JIT interference, but the generated code is slower than that produced by the JIT and therefore—because the mutator is running slower—does not stress the garbage collector as much.

Table 5.2: Absolute performance for full-heap collector.

Benchmark	Trigger (MB)	Time (s)			Memory (MB)		MMU
		Total	Mutator	Collector	Peak	Average	
compress	24	8.99	8.16	0.13	28.77	14.45	70%
jess	8	8.16	6.53	1.64	12.16	8.20	69%
raytrace	16	4.50	3.43	1.07	29.28	19.98	69%
db	24	13.18	12.38	0.80	32.62	20.17	67%
javac	24	6.37	4.99	1.38	49.27	32.78	67%
mpegaudio	8	10.24	10.24	0.00	2.47	2.41	100%
mtrt	24	3.13	2.39	0.74	82.97	46.87	69%
jack	8	4.22	3.63	0.59	10.48	6.90	69%
antlr	20	5.43	5.06	0.36	23.64	14.26	69%
bloat	24	30.83	26.99	3.83	45.62	20.24	69%
chart	36	159.80	147.50	12.24	51.14	25.55	67%
eclipse	64	90.14	77.47	12.67	80.86	66.66	56%
fop	24	3.21	2.86	0.35	27.22	22.09	70%
hsqldb	144	4.75	4.30	0.45	158.48	116.29	70%
jython	20	22.44	18.53	3.91	46.72	24.89	67%
luindex	20	17.71	16.59	1.12	21.38	14.86	68%
lusearch	36	17.29	13.18	4.11	48.75	34.79	68%
pmd	48	30.34	24.98	5.36	71.30	47.00	68%
xalan	128	12.49	11.43	1.05	136.86	87.49	64%

5.3.1 Generational versus Non-Generational Comparison

We performed a comparison of the generational and non-generational Metronome systems using the SPECjvm98 [SPEC, 1999] and DaCapo [Blackburn et al., 2006] benchmark suites. A summary of the results is shown in Tables 5.2 and 5.3. Table 5.2 gives absolute figures for the non-generational system, including the mature collection trigger, a breakdown of execution times, memory usage, and achieved minimum mutator utilization. Table 5.3 then shows results for generational Metronome (normalized against the non-generational system to facilitate comparison) with two addition columns showing the nursery trigger used, and the total fraction of collection time spent collecting the nursery. For both systems the mature trigger was held constant, and the minimum mutator utilization target specified was 70% utilization in each 10ms window.

The full heap triggers are based on each program's steady-state allocation rate and maximum live memory size; the nursery trigger was selected by evaluating a range of possibilities (512KB through 16MB) and picking the trigger that enabled the best time/space performance. Note that these are *triggers* and not *heap sizes*.

[2]In addition to adding generational capabilities, support for the Real-Time Specification for Java (RTSJ) standard [Bollella and Gosling, 2000] was disabled, and defragmentation was enabled for both the base and generational configurations of the JVM. This means that performance results for our base system can not be directly compared to that of the IBM product.

[3]Available from ftp://linuxpatch.ncsa.uiuc.edu/rt-linux/rhel4u2/R1/rtlinux-src-2006-08-30-r541.tar.bz2

Table 5.3: Generational Metronome performance relative to full-heap collector.

Benchmark	**Trigger** (MB)		**Time** (s)			**Nursery** (%)	**Memory** (MB)		**MMU**
	Mature	Nursery	Total	Mutator	Collector	of Collector Time	Peak	Average	
compress	24	2	0.99	0.98	1.87	84%	1.00	1.01	69%
jess	8	2	0.84	0.94	0.43	77%	0.69	0.80	69%
raytrace	16	2	0.76	0.90	0.28	81%	0.99	0.55	70%
db	24	2	1.00	1.00	0.89	57%	1.09	1.04	68%
javac	24	2	1.14	1.09	1.35	92%	1.70	2.03	68%
mpegaudio	8	2	1.01	1.01	1.00	—	0.78	0.77	100%
mtrt	24	2	0.88	0.97	0.61	75%	0.93	0.55	67%
jack	8	2	0.92	0.97	0.64	81%	0.82	0.90	70%
antlr	20	4	0.94	0.91	1.33	59%	1.03	1.04	68%
bloat	24	4	0.88	0.94	0.47	92%	0.56	0.83	69%
chart	36	4	0.99	1.06	0.24	80%	0.80	1.10	67%
eclipse	64	8	0.95	0.98	0.78	65%	1.23	0.75	67%
fop	24	4	1.00	1.00	0.94	82%	0.89	0.83	69%
hsqldb	144	16	1.47	1.24	3.71	100%	1.11	0.89	63%
jython	20	4	0.93	0.99	0.69	69%	0.76	0.68	63%
luindex	20	4	1.06	1.03	1.41	79%	1.02	1.04	69%
lusearch	36	8	0.97	1.00	0.88	35%	1.11	0.98	66%
pmd	48	4	0.98	0.88	1.42	89%	2.48	1.68	66%
xalan	128	12	1.16	1.11	1.80	71%	1.00	1.04	68%
Geometric Mean			**0.983**	**0.997**	**0.883**		**0.995**	**0.930**	

Because of the nature of incremental collection, for a given set of parameters the system might require differing amounts of memory to run without violating real-time requirements. When comparing stop-the-world collectors, a simpler methodology may be used in which the heap size is fixed and the resulting throughput is measured. With a real-time collector there is an additional degree of freedom, so the comparison is more complex, with an inter-relationship between total run time, total memory usage, and MMU. For each benchmark, the first column reports the full heap and nursery triggers used for that benchmark.

The reported memory size is inclusive of both mature and nursery memory. This allows a fair comparison and it reflects the nature of our system, in which nursery pages and heap pages are intermingled in physical memory. Note that the full heap collection trigger is with respect to this total usage, *inclusive* of memory consumed by the nursery.

As predicted by the model presented in Section 5.2, generational collection is better for many, but not all benchmarks. Overall, it reduces both time and space requirements, with most time improvements coming from a reduction in collection time. However, time varies from a 24% speedup for raytrace to a 47% slowdown for hsqldb, and space varies from a 44% reduction for bloat to a 148% increase for hsqldb. Real-time performance (MMU) is essentially the same, with the largest variation being 7% degradation for hsqldb. Many benchmarks have short periods during

which they exhibit non-generational behavior, leading to peak memory usage higher than the non-generational system, while average usage across the whole execution is lower. An example is eclipse, where the generational system has a peak usage 25% higher, but average memory use is just 75% of the base system over the entire run. Overall, for programs that are at least somewhat generational in their memory allocation and usage patterns, the generational collector offered significant performance benefits. Our worst result is a 47% slowdown is for hsqldb, the least generational benchmark that was tested; Blackburn et al. [2006] reported hsqldb as having a 63.4% 4MB nursery survival rate, compared with a geometric mean of 8.4% and 8.7% for the DaCapo and SPECjvm98 suites respectively.

5.3.2 Dynamic Nursery Size

The use of a single pool of pages for both the nursery and the heap, and the ability of the nursery to temporarily consume more than its trigger size, allows our collector to gracefully handle temporary spikes in the allocation rate. Table 5.4 shows the minimum, mean, and maximum nursery sizes for each benchmark (mpegaudio performs so little allocation that it never fills a 2MB nursery, so there is no data for it). Many of the benchmarks do in fact have a maximum nursery size three or more times as large as the nursery trigger, and in the case of the multithreaded mtrt benchmark, the nursery can grow to nearly 16$\times$ the trigger size. As the nursery trigger increases this effect is less dramatic, but can still be seen to some degree on most of the benchmarks. These peaks, when compared to the low average sizes, demonstrate the effectiveness of the elastic nursery size at absorbing short-term allocation spikes.

5.3.3 Parametrization Studies

Section 5.2 analytically described the effect of varying the nursery size on total memory consumption. Figure 5.8 shows the overall performance of the jess benchmark as the nursery trigger is varied from 256KB to 3MB. The example uses jess because it is known to be highly generational, allowing us to clearly see the effect of altering the nursery trigger; non-generational programs are likely to perform poorly on all feasible nursery sizes. Both the time and space measurements are point-wise normalized against the non-generational system. The most dramatic effect is that at low nursery triggers the memory usage spikes upwards (beyond the range of the graph), as predicted by the divergence condition in Eq. (5.7). At a nursery trigger of approximately 512KB, the memory consumption of the generational and non-generational systems are similar. Above 1.5MB, further increases in the nursery trigger do not improve the efficiency of nursery collections, and the space overhead from the $3N$ term begins to dominate, causing memory consumption to increase. Note that total time spent in nursery collections decreases as the nursery trigger is increased, since the total amount of data that must be promoted also decreases according to the change in $\eta(N)$. Mutator time is reasonably consistent for different nursery trigger values; changes in collector time performance correspond to (smaller) changes in

Table 5.4: Actual nursery size statistics showing dynamic nursery size variation.

Benchmark	Trigger (MB)	Average (MB)	Average ×Trigger	Maximum (MB)	Maximum ×Trigger	StdDev (MB)
compress	2	5.3	2.65	6.0	3.00	1.37
jess	2	2.0	1.00	2.4	1.20	0.02
raytrace	2	2.2	1.10	9.6	4.80	0.85
db	2	2.2	1.10	6.2	3.10	0.67
javac	2	2.9	1.45	7.1	3.55	1.31
mpegaudio	2	—	—	—	—	—
mtrt	2	2.8	1.40	31.7	15.85	3.96
jack	2	2.0	1.00	2.1	1.05	0.01
antlr	4	4.1	1.01	4.3	1.08	0.05
bloat	4	4.1	1.01	4.3	1.08	0.02
chart	4	4.1	1.01	4.4	1.10	0.04
eclipse	8	8.0	1.01	9.0	1.12	0.07
fop	4	4.1	1.01	4.2	1.04	0.04
hsqldb	16	25.3	1.58	40.0	2.50	8.45
jython	4	4.1	1.03	5.5	1.37	0.20
luindex	4	4.0	1.01	4.1	1.02	<0.01
lusearch	8	8.1	1.01	8.4	1.05	0.07
pmd	4	5.3	1.32	17.4	4.35	2.61
xalan	12	12.0	1.00	12.1	1.00	<0.01
Arithmetic Mean			**1.21**		**2.74**	

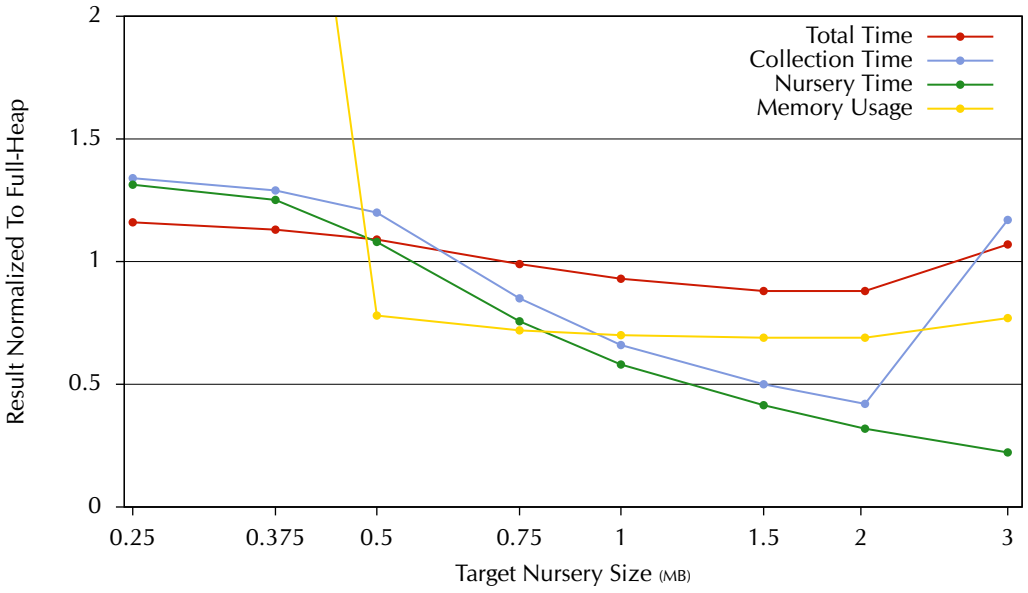
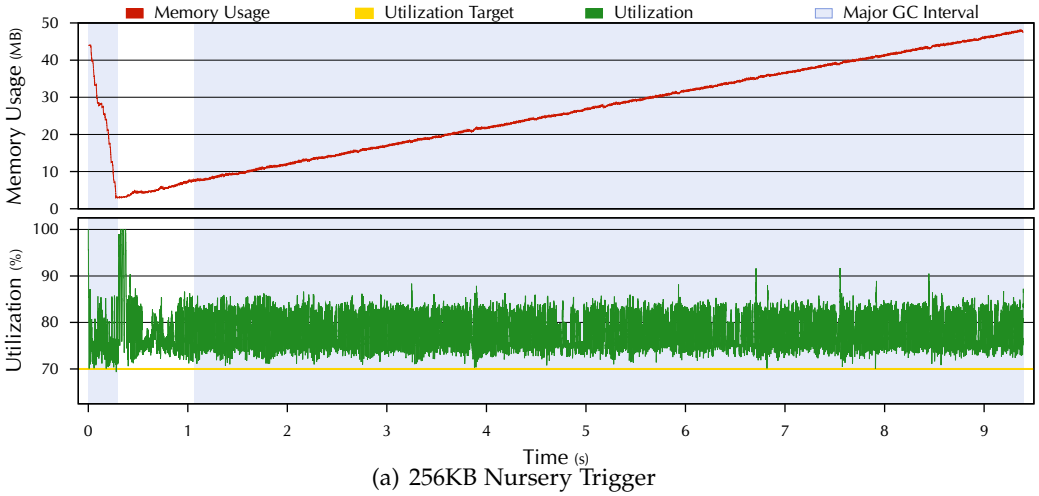
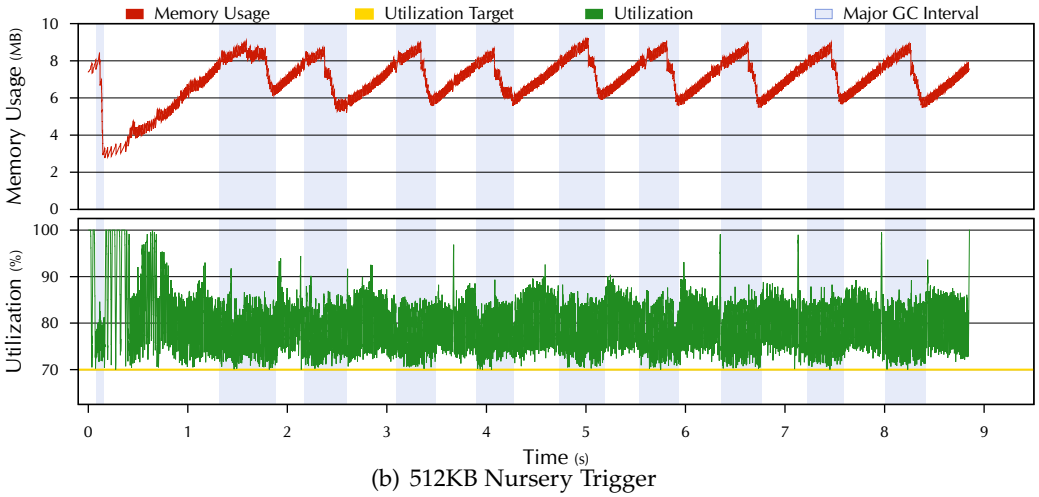


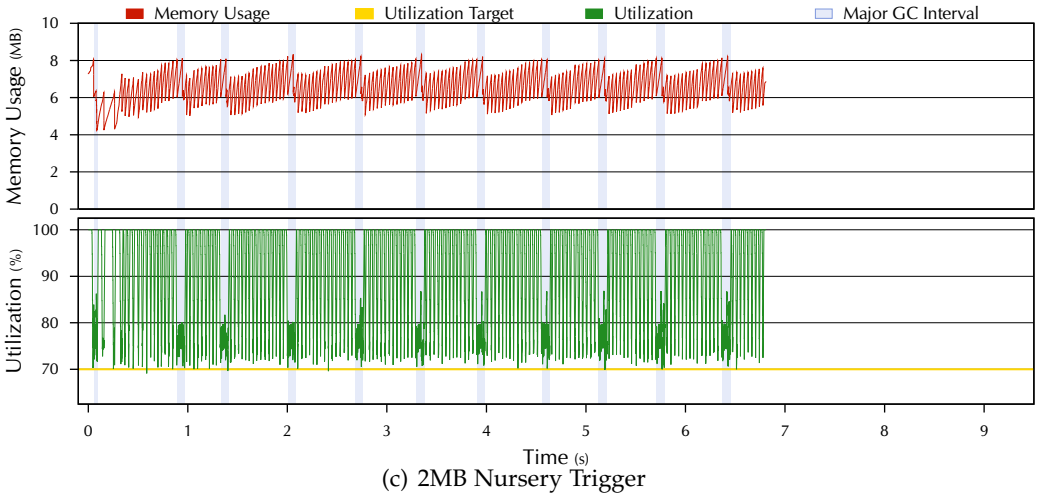
Figure 5.8: Effect of changing nursery trigger for jess with an 8MB mature collection trigger.



(a) 256KB Nursery Trigger



(b) 512KB Nursery Trigger



(c) 2MB Nursery Trigger

Figure 5.9: Performance of jess with varying nursery trigger.

overall performance.

Figure 5.9 shows the dynamic behavior of memory consumption and mutator utilization of the jess benchmark when the nursery trigger is set to 3 different values. Generally, as the nursery trigger increases, the overall efficiency of collection improves and total time spent in garbage collection decreases. For the low nursery trigger of 256KB in Figure 5.9(a), all the available collection time is spent in nursery collections, starving the mature collection and preventing it from making progress. Consequently, overall memory consumption is unbounded. The thick band shows that the utilization is always oscillating between 72% and 85%, indicating that the collector has little breathing room at all to satisfy the 70% MMU target. When the nursery trigger size is doubled to 512KB, as in Figure 5.9(b), the nursery collections complete before the subsequent nursery is filled, allowing mature collection work to occur and leading to a bounded heap size of around 9MB. While sufficient time is available to complete mature collections, mature collection cycles take approximately half a second, and utilization is still kept reasonably low and only occasionally rises to approximately 90%. When nursery size is further increased to 2MB, as in Figure 5.9(c), nursery collections complete early enough to provide a large fraction of overall collection time for mature collection, allowing major collections to complete in around a tenth of a second. With a 2MB nursery trigger, there are often periods in which neither the nursery or mature collector is active, allowing utilization to sit at 100%, with an average of around 85%. Because overall efficiency is improved, the heap consumption is lower at only 8.25MB, even though the amount of memory used by the nursery is larger.

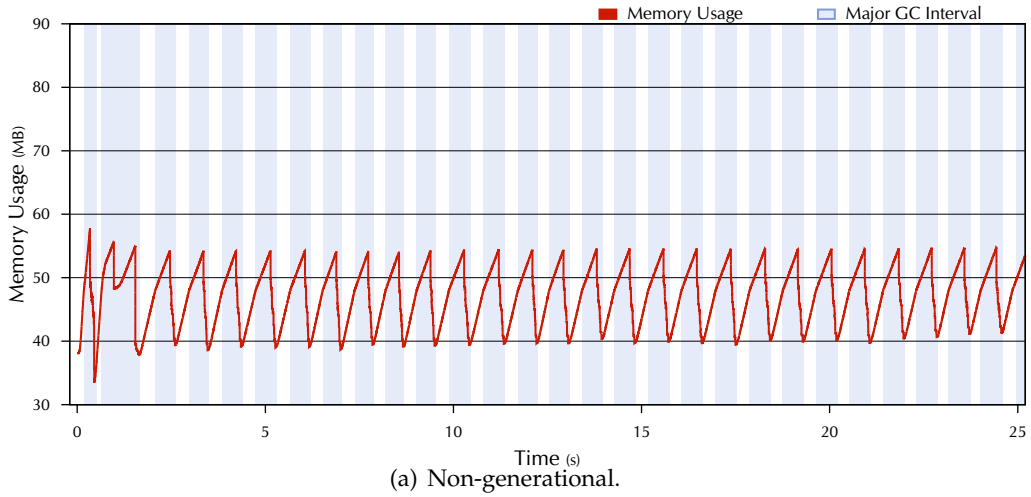
5.3.4 Start-up versus Steady State Behavior

Figure 5.10 shows the memory consumption of pjbb2000 for both systems. This benchmark begins by setting up several large data structures (the setup phase) and then runs many transactions, each of which slightly modify the pre-existing data structures (the steady-state phase). In the setup phase, both the allocation rate and the survival rate are high, forcing the generational system to grow the nursery. In this phase, the generational system uses 45% more memory than that of the non-generational system. However, once the application reaches the mission phase (about 1.8 seconds into the run), the greater efficiency of the generational system dominates, resulting in a 10% reduction in space consumption and reduced collection time.

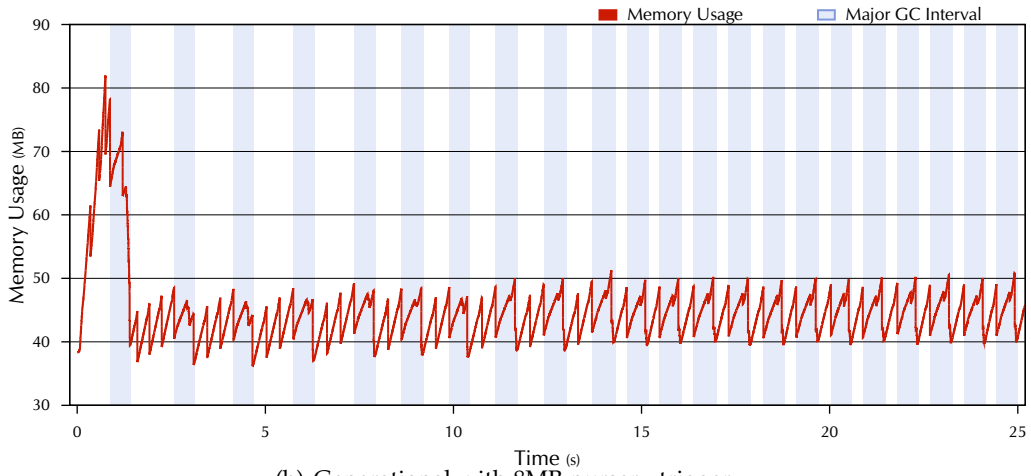
5.4 Summary

This chapter presented a new algorithm for performing generational collection incrementally in real-time, based on a three-part nursery which overlaps allocation, collection, and defragmentation. Nursery collection can be interleaved with incremental real-time collection of the mature space at any point. The resulting algorithm allows the use of large nurseries that lead to low survival rates, and yet is capable of achieving sub-millisecond latencies and high worst-case utilization.

We have implemented this new algorithm in a product-based real-time Java virtual machine, and evaluated analytically and experimentally the situations under which our generational collector is superior to a non-generational real-time collector. Programs that exhibit inherently non-generational behavior, or whose setup phase includes unusually high survival and allocation rates, will require more space to achieve corresponding real-time bounds. However, the results show that for most programs, generational collection achieves comparable real-time bounds while leading to an improvement in space consumption, throughput, or both.



(a) Non-generational.



(b) Generational with 8MB nursery trigger.

Figure 5.10: Memory usage over time of pjbb2000 under generational and non-generational collection.

Chapter 6

High-level Low-level Programming

Previous chapters presented work to address *runtime* support for high-level low-level programming, in particular through the development of garbage collectors suitable for low-level programs. This chapter looks at a separate, but equally important problem: *language* support for high-level low-level programming. That is, *how* to write low-level programs in a high-level language. While the power of high-level languages lies in the ability to abstract over hardware and software complexity, opaque abstractions are often show-stoppers for low-level programmers, forcing them to either break the abstractions, or more often, simply give up and use a different language. This chapter describes an approach to address the challenge of opening up a high-level language to practical low-level programming, *without forsaking integrity or performance*.

This chapter is structured as follows. Sections 6.2 and 6.1 expand on the definitions used in this thesis to describe low-level programming, and to differentiate high- and low-level programming languages. Section 6.3 looks at previous shifts in language-use of low-level programming, and Section 6.4 relates these previous changes to our current environment. Having motivated the goal of high-level low-level programming, Section 6.5 then provides an overview of previous work that has combined high-level languages and low-level programming, giving a categorization of these techniques, and identifying the work that serves as the foundation on which the techniques described in the following chapter are built.

This chapter describes work published in the paper "Demystifying Magic: High-level Low-level Programming" [Frampton, Blackburn, Cheng, Garner, Grove, Moss, and Salishev, 2009a].

6.1 Low-level Programming

Recall from the introduction that high-level low-level programming was defined as that which requires transparent, efficient access to the underlying hardware and/or operating system. Low-level software includes operating systems, virtual machine runtimes, hardware device drivers, software for embedded devices, and software for real-time systems. Note that the definition excludes some programs that fall under the broader umbrella of *systems* programming. Compilers, for example are a key

example of systems programs that often do not require any low-level programming.

Low-level programs place unique requirements on both a language and its execution environment. A common requirement is a greater degree of control over low-level details, such as memory layout and machine register usage, or the ability to access hardware-specific features such as special machine instructions. These requirements often arise as the low-level software may need to conform to externally defined interfaces, such as those required by a host operating system or hardware devices. It may also be necessary to control interactions with cache and memory for both correctness and performance reasons. Because low-level software is often the foundation on which other software is built, *throughput*, *responsiveness* and *deterministic performance* may be critical concerns. It is also essential that the programmer be able to develop an intuitive model of how the code will execute, in order to facilitate optimization and debugging. These requirements are very broad, and are not universally required by every low-level program, but—in order to support low-level programming in general—it must be possible to deliver on all these requirements, both together and in isolation.

6.2 High- versus Low-level Languages

Recall that throughout this thesis, we define *high-level languages* as those languages that are type-safe, memory-safe, provide encapsulation, and strong abstractions over hardware (e.g., Java and C#). In contrast, low-level languages (e.g., C) are those languages that provide a transparent view onto the hardware, and do not provide rich runtime services. While C++ [Stroustrup, 1986] includes some higher-level language features, we class it as a low-level language in this work due to the absence of strong type- and memory-safety, as well as the lack of enforced abstraction from low-level detail.

Our definition does not discount the potential benefits of other high-level languages for low-level programming. The use of the described high-level languages for low-level programming has both a well demonstrated potential [Alpern et al., 1999; Blackburn et al., 2004b; Hunt et al., 2005; Garner et al., 2007], as well as identified and significant roadblocks [Shapiro, 2006] that need to be addressed to drive future adoption.

6.3 From Assembler to C

The earliest computers ran without any operating system: individual jobs including program and data were fed into machines and executed. In time, pre-loaded runtime libraries were provided, which later evolved into complete operating systems. Note that until recently, high-level languages were considered to include all languages that abstracted over the level of assembly language [Sammet, 1969], and included many languages which we now consider low-level (e.g., BLISS). Sammet [1971] provides an

overview of the early usage of these higher-level languages—as opposed to assembly language—for systems implementation. Quoting Sammet [1972]:

> The use of higher level languages for systems programming has finally been recognized as being both legitimate and practical. After almost 15 years of debate and negative views, an increasing number of systems programs are actually being written using higher level languages...

There were several early and successful systems implementations using higher-than-assembly-level languages in the 1960s and into the early 70s, including the operating systems for Burroughs computers which were written using a hierarchy of extended ALGOL languages [Lyle, 1971]. Graham [1970] reflects on the choice of language through his experience with Multics [Corbató and Vyssotsky, 1966] which was written almost entirely in IBM PL/I. Other languages included extensions to common languages of the time such as Pascal and FORTRAN, as well as custom systems programming languages such as BLISS [Wulf et al., 1971a,b].

History has judged the most significant development in systems implementation language from this era to be C [Kernighan and Ritchie, 1988], which evolved from BCPL and B and was the language upon which UNIX was built in the early 1970s. Today, some 30 years later, C and C++ remain the dominant languages for systems implementation.

6.3.1 Complexity Drives Change

The change from assembly to C was largely driven by the increasing complexity of both *hardware platforms* and *software requirements*. Both of these trends have continued, and one need only compare the hardware in use today to that of early computers, or the complexity of a modern operating system to the first operating systems, to understand why assembly language is no longer seriously considered a reasonable implementation language (although assembly is still used for small sections of code).

Landau [1976] shows one way early hardware evolution contributed to the move away from assembly language. When optimizing several multi-instruction calculations on a super-scalar processor, assembly programming leads to either inefficient or obfuscated code. Ordering the instructions to maximize readability will leave parts of the hardware idle, while optimizing for performance mixes together the parts of each individual computation, making the code harder to read and understand. While there are hardware designs that alleviate this particular issue through out-of-order execution, the general principle that a level of abstraction provides an opportunity for optimization—without affecting the readability of the source—remains.

6.3.2 Cultural Resistance

Today it is generally considered uncontroversial to implement low-level code in languages such as C. This was not always the case, and Sammet [1971] notes the significant cultural resistance to the shift from assembly programming to the adoption of higher-level languages:

> There are many factors which go into choosing whether or not to use a higher level language. Unfortunately, one of the major ones affecting a decision still is the snob reaction of the systems programmers who feel that such aids are good enough for the applications programmers but not good enough for them.

This position persisted for some time, and is demonstrated by Fletcher [1975], who spoke against the use of higher-level languages (emphasis added):

> Our experience also contrasts with many of the assertions made about assembly language system programs: We do not find that they are more prone to bugs, are less clear or less well organized (structured), require a greater investment in time or money, or are less easy to modify. *We can only suppose that findings to the contrary are saying something about the skills of the programmers involved rather than about languages.*

6.4 Looking Forward

There is, of course, cultural resistance to a change away from C for low-level programming. While this may prove a significant barrier to adoption, we believe that once the *practical* issues are understood and addressed, further increases in complexity will inevitably drive the adoption of higher level languages, just as they drove the transition from assembly to C.

While C is still the dominant language of choice for low-level programming, the same can not be said regarding language usage for general application software. The TIOBE Index [TIOBE, 2009] shows Java—a language that did not exist prior to 1995—as the single most popular language. As software becomes more central to our lives, there is a demand for greater security and reliability—for safety, economic and privacy reasons—as well as a need for increased programmer productivity.

Increases in hardware complexity appear to be accelerating. As the ability to improve performance using traditional processor designs evaporates [Agarwal et al., 2000], hardware vendors are increasingly turning to multi- and many-core designs. We are also seeing an increase in heterogeneous core systems, such as the synergistic processing elements of the Cell processor [Kahle et al., 2005] , the use of graphics processing units for general purpose computation [Garland et al., 2008; Tarditi et al., 2005], and the use of FPGAs to fabricate custom processors at run-time [Huang et al., 2008]. There are also more radical designs for next generation processors such as the EDGE architecture [Burger et al., 2004] with a programmable grid of processing units with non-uniform access to cache and memory.

New processor designs, increasing levels of concurrency, and the addition of dynamic power-saving features—such as scaling or shutting down various cores—continue to increase complexity, and provide new challenges that affect how we must write and run software.

High-level language abstractions are a well understood tool for helping to meet these requirements. By lifting the level of abstraction from machine-specific details to the level at which the problem is best understood, it is possible to provide increases in productivity. Abstractions also help to address security and reliability issues by providing features such as type and memory safety—which make some classes of programmer error impossible—as well as creating an environment that facilitates automated analysis and verification.

Retrospectives of C [Ritchie, 1993] and C++ [Stroustrup, 1993, 2007] indicate how the respective languages might be designed differently in a more modern environment, if starting from scratch without the constraint of maintaining compatibility. While identifying the benefits of garbage collection and advocating its future inclusion in C++, Stroustrup [2007] notes that '...C++ would have been stillborn had it relied on garbage collection when it was first designed'. As a community we must continue to evolve our development methodologies and languages alongside changes in hardware, software requirements, and our increased understanding of programming language design and software engineering.

6.5 Related Work

Given the potential benefits of high-level languages, it is unsurprising that many have sought to combine low-level programming and high-level languages. This section outlines the main techniques that have been employed: introducing new languages, fortifying low-level languages, using two languages, and extending high-level languages. The following two key observations help to categorize the various approaches to combining high-level languages and low-level programming:

Observation 1 *High-level languages provide abstractions that can lead to software with greater security, better reliability, and lower development costs.*

Observation 2 *The goals of high-level languages tend to be at odds with low-level programming [Shapiro, 2006]. This is primarily because high-level languages gain power through abstracting over detail, while low-level programming may require transparent access to detail.*

There are other concerns related to performance, determinism, portability, and existing programmer skills—among others—but they lie largely outside the scope of this discussion.

6.5.1 Fortifying a Low-level Language

While individual motivations vary, it is clear that systems programming projects have found it desirable to reach for higher levels of abstraction akin to those found in high-level languages. This has a strong history stretching back to the 1970s [Fletcher, 1975; Fletcher et al., 1972; Frailey, 1975; Horning, 1975].

In today's context, memory safety is an area in which C is conspicuously lacking, and there have been countless idioms and techniques devised to improve the situation including conservative garbage collection [Boehm, 1993; Boehm and Weiser, 1988], smart pointers and reference counts [Gay et al., 2007], static and dynamic analysis tools such as Valgrind [Nethercote and Seward, 2007], as well as custom allocators such as `talloc` [Tridgell, 2004]. This fortification process also feeds into revisions of language specifications; future versions of the C++ specification are expected to include both high-level language features (e.g., garbage collection) as well as additional systems programming features (e.g., `constexpr` functions that are resolved at compile time).

The SAMBA Developers Guide [Vernooij, 2008] includes a set of *coding suggestions* which amount to a list of conventions and idioms designed to work around language shortcomings in areas such as safety and portability. Enforced by convention rather than by the semantics of the underlying language, these often exist to simply work around artificial limitations of the base language.

There is also a limit to the extent one can introduce high-level abstractions into an existing language. High-level abstractions such as threads have been shown to be problematic to implement [Boehm, 2005], since correctness can not be ensured without the cooperation of the underlying language.

6.5.2 Systems Programming Languages

There has been a long history of language development targeting low-level or systems implementation, with differing degrees of innovation and success [Cardelli et al., 1989; Kernighan and Ritchie, 1988; Richards, 1969; Stroustrup, 1986; Wulf et al., 1971b]. The Modula series of languages [Cardelli et al., 1989] was notable in several respects. It had a stronger notion of type safety, integrating garbage collection into the programming environment by providing two heaps for dynamic allocation: one garbage collected and the other explicitly managed. Modula-3 also included a *module system*, now uniformly considered an important building block for developing large and complex systems. The lack of a module system to provide intermediate levels of visibility is noted as a shortcoming in C [Ritchie, 1993], and—prior to the introduction of namespaces in 1994—also in C++ [Stroustrup, 1993].

There have also been attempts at creating safer derivatives of C, an example being Cyclone [Jim et al., 2002], which introduces stronger type and memory safety, as well as a safe approach to multithreading [Grossman, 2003].

This approach—in addition to approaches that aim to provide richer static analysis tools to prove 'unsafe' code correct [Ferrara et al., 2008]—focuses on proving low-level programming techniques correct, rather than allowing high-level low-level programming. We believe this neglects our second observation above: by maintaining a transparent view to low-level details *across the board*, much of the potential gains of high-level abstractions are lost.

6.5.3 Two-Language Approaches

The most common technique used to resolve the tension between high-level and low-level programming is to simply use different languages or dialects for each task. This general approach affords itself to several solutions, based primarily around the level of integration of the two languages.

Extreme: Do not use high-level languages for low-level tasks. This is perhaps the most extreme position but is also the status quo, demonstrated through the continued dominance of C and C++ for low-level programming while other languages continue to enjoy increased popularity for general programming tasks.

Intermediate: Call out using a Foreign Function Interface (FFI). This technique provides an *escape hatch* where the programmer can call into a separate language to implement low-level features, and is available in almost all modern language environments, from C's ability to call into assembly, to Java [Gosling et al., 2005] with the Java Native Interface [Liang, 1999] and C# [Ecma, 2006a] (in the Common Language Infrastructure [Ecma, 2006b]) with Platform Invoke. This allows some low-level programming, such as that in Java described by Ritchie [1997], but it is a coarse approach and the split between low-level and high-level code can compromise both software design and performance.

Minimal: Introduce dual-semantics to the language. A refinement of the FFI technique is to introduce regions within the high-level language that allow the use of low-level language features. This allows greater coherency between the high- and low-level aspects of the system being implemented. Modula-3 [Cardelli et al., 1989] achieves this through *unsafe modules* (where features such as the use of pointers or unchecked type casts are allowed) and *unsafe interfaces* (which can only be called from unsafe modules). Safe modules may interact with unsafe modules by exposing an unsafe module through a safe interface. C# [Ecma, 2006a] and the CLI [Ecma, 2006b] also use a similar concept of unsafe code, but control the the interaction of safe and unsafe code at a coarser granularity.

Techniques to reduce both the design [Hirzel and Grimm, 2007] and performance [Stepanian et al., 2005] disadvantages of these approaches exist. There is however a more fundamental problem in that they treat the need to perform low-level operations as an all-or-nothing requirement. This does not resolve well with our above observations: it should be possible to leverage high-level abstractions for everything other than the *specific* low-level detail you are dealing with.

6.5.4 Extending High-level Languages for Low-level Programming

Another approach is to provide the tools with which one can *extend* a high-level language to perform low-level programming. This is the general approach taken

throughout the work in the following chapter, and this section describes the foundations on which that work is built. Much of the progress in extension for low-level programming has been through projects where the focus was not on language design itself. Two areas of research in particular—operating systems and virtual machines—have been largely responsible for the progress to date, although their contributions to language design have generally been driven by other pragmatic goals.

Operating systems have been developed using high-level languages including Modula-3 [Bershad et al., 1994], Haskell [Hallgren et al., 2005], Java [Prangsma, 2005], and C# [Hunt et al., 2005]. SPIN [Bershad et al., 1994] is a research operating system focused on safety, flexibility, and extensibility [Bershad et al., 1995], and is written in an extended version of Modula-3 [Cardelli et al., 1989]. Extensions include improvements to allow interoperability with externally defined interfaces (such as for accessing hardware devices), changes to improve error-handling behavior, and the ability to safely cast raw memory into typed data [Fiuczynski et al., 1997]. Yamauchi and Wolczko [2006] embed a small virtual machine within a traditional operating system to allow safer drivers written in Java, while the Singularity project [Hunt et al., 2005; Hunt and Larus, 2007] (written using Sing#, an extension of Spec#, itself an extension of C#) aims to discover how a system should be built from the ground up for high-level language execution, including models for inter-process communication [Aiken et al., 2006; Fähndrich et al., 2006].

There have been many examples of virtual machines written using high-level languages [Alpern et al., 2000; Blackburn et al., 2008; Flack et al., 2003; Rigo and Pedroni, 2006; Simon et al., 2006; Sun; Ungar et al., 2005; Whaley, 2003], most likely due to the combination of a systems programming task in concert with a deep understanding of a high-level language. Virtual machine development is the context within which much of our work has been undertaken. Jikes RVM, formerly known as Jalapeño [Alpern et al., 2000] is a high-performance Java-in-Java virtual machine, requiring extensions—known as *magic*—to support required low-level operations [Alpern et al., 1999]. Maessen et al. [2001] provided a deeper understanding of how magic operations interact with the compiler, and what steps must be taken to ensure correctness in the face of compiler optimizations. OVM, a Java-in-Java virtual machine designed for real-time applications, uses similar magic idioms, but has built more principled abstractions around them [Flack et al., 2003]. Moxie [Blackburn et al., 2008] is a clean-slate Java-in-Java virtual machine that was used to prototype some of the ideas that have helped to feed into our approach. The `sun.misc.Unsafe` API, implemented by current production virtual machines, and implemented in Jikes RVM through our more general magic framework, provides some similar functionality. Interestingly, it may be possible to use `sun.misc.Unsafe` as a pragmatic means to implement a limited subset of our framework on existing production virtual machines.

6.6 Summary

This chapter discussed the 1970s shift in the accepted language for low-level systems implementation from assembler to languages such as C, and identified increases in hardware and software complexity, alongside improvements in language technology, as key drivers behind this change. Relating this historical shift in the accepted low-level implementation language to our current environment serves to further motivate our goal of high-level low-level programming. After enumerating various approaches to combining low-level programming with high-level languages, this chapter identified language extension as the approach with the most promise. This approach forms the basis for the work described in the following chapter.

Chapter 7

High-level Low-level Programming with `org.vmmagic`

The previous chapter motivated the goal of high-level low-level programming. This chapter shows how this goal can be achieved, introducing an abstract approach and a concrete framework, `org.vmmagic`, that allows high-level low-level programming in Java.

This chapter is structured around two key sections: Section 7.1, which describes the approach to high-level low-level programming; and Section 7.2, which describes the concrete framework, `org.vmmagic`, constructed using the approach. Section 7.3 then gives a brief overview of the status of the `org.vmmagic` framework.

This chapter describes work published in the paper "Demystifying Magic: High-level Low-level Programming" [Frampton, Blackburn, Cheng, Garner, Grove, Moss, and Salishev, 2009a].

7.1 The Approach

This section describes our approach to low-level programming in a high-level language. The premise of this discussion is that *high-level programming is desirable whenever it is reasonably achievable*. High-level languages are designed to abstract over the specifics of the target environment, shielding the programmer from complexity and irrelevant detail so that they may focus on the task at hand. In a systems programming task, however, there is often a need for transparent access to the lowest levels of the target environment. The presence of high-level abstractions can obstruct the programmer in this objective.

7.1.1 Key Principles

Our approach is guided by a principle of containment, whereby we minimize exposure to low-level coding both in *extent* (the number of lines of code) and *severity* (the degree to which semantics are altered). Our view is that to achieve this efficiently, effectively, and safely, adding low-level features to high-level languages requires: (1)

extensibility, (2) encapsulation, and (3) fine grained divergence. The following paragraphs describe each of these attributes in more detail.

Extensibility. To reach beyond the semantics of a high-level language, systems programmers need to be able to either change the language (generally infeasible), use a different language (undesirable), or extend the language. Jikes RVM took the third approach. However, the original Jikes RVM approach had two notable shortcomings: a) the extensions were unstructured, comprising a potpourri of ad hoc extensions accreted over time; and b) the extensions required modification to the compiler(s) and runtime. An extensible framework for introducing and structuring low-level primitives is necessary. Such a framework will maximize reuse and not require modifying the source of the language runtime in order to provide new extensions. The extensible framework is discussed in Section 7.2.

Encapsulation. Thorough containment of low-level code is essential to minimize the erosion of any advantages of the high-level language setting. Two-level solutions, such as those provided by foreign function interfaces (FFIs) [Liang, 1999], unsafe sublanguages [Cardelli et al., 1989; Ecma, 2006a], or other means [Hirzel and Grimm, 2007], tend to polarize what is otherwise a rich spectrum of low-level requirements. Consider the implementation of a managed runtime, where on one hand the object model may internally require the use of pointer arithmetic, while the scheduler may instead require low-level locking and scheduling controls. Simply classifying both as 'unsafe' renders both contexts as equivalent, reducing them to the same rules and exposing them to the same pitfalls. By contrast, a general mechanism for *semantic regimes* may allow low-level code to be accurately scoped and encapsulated, avoiding under- or over-specification.

Encapsulation is illustrated in Figure 7.1, where a safe method, `getHeader()`, is implemented through safe use of an unsafe memory operation, `loadWord()`.[1] The `@UncheckedMemoryAccess` annotation is used to scope the method to explicitly permit its use of `loadWord()`, while the `@AssertSafe` annotation encapsulates the unsafe code by asserting that calls to `getHeader()` are 'safe'. This allows `getHeader()` to be called from any context. The result is a more general and extensible means of describing and encapsulating low-level behavior than the practice of simply declaring entire contexts to be either 'safe' or 'unsafe'. The implementation of semantic regimes in our concrete framework is discussed in Section 7.2.2.2.

Fine grained divergence. A key issue when altering semantics is the granularity at which that divergence occurs with respect to program scope. Coarse grained approaches, such as the use of FFIs, suffer both in performance and semantics. Performance suffers because of the impedance mismatch between the two language domains. In some cases, crossing this boundary requires heavy-weight calling conven-

[1] The safety of `getHeader()` is due to the use of the strongly typed `ObjectReference`. The method would not have been safe had the weakly typed `Address` (i.e., **`void`**$*$) been used.

```
1  @UncheckedMemoryAccess
2  @AssertSafe
3  public Word getHeader(ObjectReference ref) {
4    return ref.loadWord(HEADER_OFFSET);
5  }
```

Figure 7.1: Unsafe code encapsulated within a safe method.

tions [Liang, 1999], and it is generally difficult or impossible for the high-level language's compiler to optimize across the boundary. (Aggressive compiler optimizations have recently been shown to reduce this source of overhead [Stepanian et al., 2005].) Similarly, the coarse grained interface can generate a semantic impedance mismatch, requiring programmers who work at the interface to grapple with two distinct languages. Instead, we argue for introducing semantic deviation at as fine a grain as possible. Thus in the example of the object model in Figure 7.1, the programmer implementing `getHeader()` must (of course) reason about the layout of objects and their headers in memory, but is not required to code in an entirely distinct language, with all the nuances and subtleties that entails. Further, an optimizing compiler can reason about `loadWord()`, and, if appropriate, inline the `getHeader()` method and further optimize within the calling context. In practice, the result yields performance similar to a macro in C, but retains all of the strengths of the high-level language except for the precise concern (memory safety) that the programmer is required to dispense with.

7.1.2 Requirements and Challenges

Having outlined our approach at a very high level, we now explore the primary concerns that face the construction of a framework for high-level low-level programming. The challenges of low-level programming in a high-level language fall broadly into two categories: 1) the high-level language does not allow data to be represented as required, and 2) the high-level language does not allow behavior that is required.

7.1.2.1 Representing Data

Low-level programming may often require types that are not available in the high-level language. For example, high-level languages typically abstract over architecture, but low-level programming may require a type that reflects the underlying architectural word width. Additionally, an operating system or other interface may expect a particular type with a certain data layout which is unsupported by the high-level language.

Primitive types. It may be necessary to introduce new primitive types—types that could otherwise not be represented in the language—such as architecturally dependent values. In the original Jalapeño, a Java **`int`** was used to represent an architec-

tural word. This suffered from a number of fairly obvious shortcomings: Java **int**s are signed, whereas addresses are unsigned; a Java **int** is 32-bits, making a 64-bit port difficult; and aliasing types is undesirable and undermines the type safety of the high-level language. (For the 64-bit port, it was necessary to disambiguate large numbers of **int**s throughout the code base, and determine whether they were really addresses or integers [Venstermans et al., 2006]). Ideally, systems programmers would be able to introduce their own primitive types for such purposes. This objective might imply that operators over those types could be added too.

Compound types. Systems programmers must sometimes use compound types to efficiently reflect externally defined data, such as an IP address. Because these are externally defined, it is essential that the programmer have precise control over the layout of the fields within the type when required. Typically, a language runtime will by default do its best to pack the fields of a type to optimize for space, or to improve performance through better locality, etc. However, the need to interface with externally defined types means that the user must be able to optionally specify the field layout. Some languages (e.g., C# [Ecma, 2006a]) provide the programmer with fine control over field layout, but others (e.g., Java [Gosling et al., 2005]), provide none.

Unboxed types. High-level languages allow users to define compound types. However, these types are often by default 'boxed'. Boxing is used to give an instance of a type its identity, typically via a header which describes the type and may include a virtual method table (thus identifying the instance's semantics). From a low-level programmer's point of view, boxing presents a number of problems, including that the box imposes a space overhead and that the box will generally prevent direct mapping of a type onto some externally provided data (thereby imposing a marshaling/copying overhead at external interfaces). *Unboxed types*—types that are stripped of their 'box'—allow programmers to create compound types similar to C `structs`. User-defined unboxed types are not uniformly supported by high-level languages (for example, Java does not offer user-defined unboxed types). Integration of unboxed types into an environment implies a variety of restrictions. For example, subtyping is generally not possible because there is no way of reestablishing the concrete subtype from the value due to the absence of a box that captures the instance's type. Furthermore, in some languages there is no way to refer to an instance of an unboxed type (if, for example, the language does not have pointers, only object references), which limits unboxed types to exist as fields in objects or as local variables. C# provides unboxed types, and supports interior pointers to unboxed types as fields of objects.

References and values. Conventionally, data may be referred to directly (by value) or indirectly (by reference). In many high-level languages, the language designers choose not to give the programmer complete freedom, preferring instead the simplic-

ity of a programming model with fixed semantics. For example, in Java, primitive types are values and objects are references; the system does not allow an object to be viewed as a value. Thus Java has no concept of pointer, and no notion of type and pointer-to-type. Since pointers are a first order concern for systems programmers, a low-level extension to a high-level language should include pointers, and allow the value/reference distinction to be made transparent when necessary.

7.1.2.2 Extending the Semantics

In the limit, a systems programmer will need to access the underlying hardware directly, unimpeded by any language abstractions. This problem is typically solved by writing such code in assembler, following a two-language approach. Our alternative is to add *intrinsic functions* to the language—which directly reflect the required semantics, and *semantic regimes*—within which certain language-imposed abstractions are suspended or altered.

Intrinsic functions. Intrinsic functions allow the addition of operations that are not expressible in the high-level language. An example of this is a systems programmer's need to control the hardware caches. For example, Jikes RVM (like most virtual machines) dynamically generates code and for correctness on the PowerPC platform, must flush the data cache and invalidate the instruction cache whenever new code is produced. However, a high-level language such as Java abstracts over all such concerns, so a programmer would typically resort to a two-language solution. Likewise, the implementation of memory fences [Lea] and cache prefetches [Garner et al., 2007] require semantics that are architecture-specific, and that a high-level language will abstract over. Intrinsic functions are widely used, and in the case where the systems programmer *happens* to be maintaining the very runtime on which they depend, they may readily implement intrinsic functions to bypass the language's restrictions. Ideally, a high-level language would provide some means for extensible, user-defined intrinsics. In that case, the user would need to provide a specification of the required semantics. In the limit, such a specification may need to be expressed in terms of machine instructions, augmented with type information (to ensure memory safety) and semantic information (such as restricting code motion) essential to allowing safe optimization within the calling context.

Semantic regimes. In addition to *adding* new operations to the semantics of the high-level language, sometimes low-level coding will necessitate *suspending* or *modifying* some of the semantics of a high-level language. This scenario is particularly common when a virtual machine is implemented in its own language, as it must curtail certain semantics to avoid infinite regress; the virtual machine code that implements a language feature cannot itself use the language feature it implements. For example, the implementation of `new()` cannot itself contain a `new()`. For semantics that are directly expressed in the high-level source code (such as `new()`), this is achievable through careful coding. However, an explicit semantic regime can be a valuable aid

in automatically enforcing these restrictions. In other cases, the semantics that need to be suspended are not controllable from the high-level language. For example, low-level code may need to suspend array bounds checks, avoid runtime-inserted scheduling yieldpoints, be compiled to use non-standard calling conventions, or be allowed to access heap objects without executing runtime-inserted garbage collector read/write barrier sequences. By defining orthogonal and composable semantic regimes for each of these semantic changes, the programmer can write each necessary low-level operation while preserving a maximal subset of the high-level language semantics. Thus ideally a runtime would provide a means of defining new semantic regimes and applying such regimes to various programming scopes.

7.2 A Concrete Framework

This section takes the general approach outlined in the previous section and shows how it is implemented it in practice. Concretely, we introduce a framework for building language extensions that allows Java to support low-level programming features. This framework is the basis for the publicly available `org.vmmagic` package.[2] We characterize the extensions in terms of the same categories used in the preceding section: extending the type system and extending language semantics.

In addition to the requirements discussed in the previous section, for our concrete realization we added the pragmatic goal of minimizing or eliminating any changes to the high-level language syntax. This enables us to leverage existing tools and retain portability.

```
1 class Address {
2   ...
3   byte loadByte();
4   void store(byte value);
5   ...
6 }
```

Figure 7.2: First attempt at an `Address` type.

To help ground the discussion, we use a running example of the evolution of an `Address` type, as shown in Figure 7.2. This is an abstraction that provides functionality similar to that provided by an untyped pointer (**`void`**$*$) in C, an unsafe feature absent from many high-level languages but essential for many low-level tasks. For simplicity, we show only a very minimal subset of the `Address` type as it evolves. Although for concreteness the example is expressed in terms of Java syntax, the abstract approach from Section 7.1.2 and many aspects of this concrete framework are language-neutral, including applicability beyond Java-like languages to others including dynamic object-oriented languages like Python.

[2] Available with Jikes RVM from http://www.jikesrvm.org.

The `org.vmmagic` package has in various forms been both used by and shaped by use in three Java-in-Java JVMs (Jikes RVM [Alpern et al., 1999], OVM [Flack et al., 2003], and Moxie [Blackburn et al., 2008]), one C/C++ JVM (DRLVM [Apache; Glew et al., 2004]), and one operating system (JNode [Prangsma, 2005]). Much of what is described here has been publicly available since the 3.0.1 release of Jikes RVM; some aspects are currently under development, and a few other clearly identified aspects of the framework are more speculative.

7.2.1 Type-System Extensions

Section 7.1.2.1 discussed the system programmer's requirement of being able to extend the type system. We address these requirements concretely through two mechanisms. The first, *raw storage*, allows the introduction of types with explicit layouts that may depend on low-level characteristics of the target system. The second allows us to introduce *unboxed* types with control over field layout.

7.2.1.1 Raw Storage

Raw storage allows the user to associate an otherwise empty type with a raw chunk of backing data of a specified size. The size may be specified in bytes, or more abstractly in terms of architectural width words (whose actual size will be platform dependent). Raw storage is a contract between the writer of the type and the runtime system which must allocate and manage the data. Raw storage is not visible to the high-level language, and can only be accessed through the use of intrinsic functions. In Figure 7.3, the `@RawStorage` annotation[3] is used to associate a single architectural word with the `Address` type.

```
1 @RawStorage(lengthInWords=true, length=1)
2 class Address {
3   ...
4   byte loadByte();
5   void storeByte(byte value);
6   ...
7 }
```

Figure 7.3: Associating a one word payload with `Address`.

This example shows how the raw storage mechanism allows systems programmers to fabricate basic (non-compound, unboxed) types. Section 7.2.2.1 discusses how the programmer can define operations over such types.

At present we have limited our framework to byte-granularity storage. However, as future work we intend to explore sub-word granularity storage and layout. Bit-grained types are important to projects such as Liquid Metal [Huang et al., 2008],

[3]Here we have used the Java annotation syntax to annotate the type. As demonstrated by Flack et al. [2003] and Alpern et al. [1999], other mechanisms such as marker interfaces can be used to similar effect when the language does not explicitly support an annotation syntax.

which have expressed an intention to use this framework. Prior work in the OVM project tentatively explored this issue [Flack et al., 2003]. The SPIN project described an example of packet filtering in Modula-3 which used bit masks and bit fields; however, the example they gave was at a 16 bit (2 byte) granularity [Fiuczynski et al., 1997].

7.2.1.2 Unboxed Types

We allow programmers to define unboxed types by marking class declarations with an `@Unboxed` annotation. Since an unboxed type is distinguished from an object only by syntax, we rely on the runtime compiler to ensure that unboxed types are never used as objects. Our current implementation in Jikes RVM is limited to supporting single field types (such as `Address`), which are treated like Java's primitives and are thus passed by value and allocated only on the stack.

Control of field layout. As Figure 7.4 shows, when specifying an unboxed type, our framework allows the programmer to specify that field order should be respected by setting the layout parameter to `Sequential`, and requires the user to pad the type with dummy fields as necessary (as is commonly done in C). This allows the programmer to precisely match externally defined types.

```
1 @Unboxed(layout=sequential)
2 class UContext {
3   UInt64 uc_flags;
4   UContextPtr uc_link;
5   StackT uc_stack;
6   ...
7 }
```

Figure 7.4: Unboxing with controlled field layout.

Support for compound unboxed types and pointers to unboxed types are not available in Jikes RVM 3.1.0, but will be released in a future version.

7.2.2 Semantic Extension

Our framework follows the discussion in Section 7.1.2.2, providing two basic mechanisms for extending the semantics of the language: 1) *intrinsic functions*, which allow the expression of semantics which are not directly expressible in the high-level language, and 2) *semantic regimes*, which allow certain static scopes to operate under a regime of altered semantics, according to some contract between the programmer and the language implementation.

7.2.2.1 Intrinsic Functions

Intrinsic functions amount to a contract between the programmer and the compiler, whereby the compiler materializes the function to reflect some agreed-upon semantics, inexpressible in the high-level language. In early implementations of magic in Jikes RVM [Alpern et al., 1999], the contract was implemented by compiler writers intercepting method calls to magic methods in the Java bytecode (identified by the class and method being called) and then realizing the required semantics in each of the three runtime compilers instead of inserting a method call.

```
1  @RawStorage(lengthInWords=true, length=1)
2  class Address {
3    ...
4    @Intrinsic(LOAD_BYTE)
5    byte loadByte()
6    ...
7    @Intrinsic(STORE_BYTE)
8    void storeByte(byte value)
9    ...
10   @Intrinsic(WORD_LT)
11   boolean LT(Address value)
12   ...
13 }
```

Figure 7.5: Use of intrinsics for `Address`.

Moxie [Blackburn et al., 2008] developed the idea further by canonicalizing semantics, separating the usage of an intrinsic operation from the semantics of the operation itself. Figure 7.5 shows how intrinsic function declarations can then reference the desired semantics, with the intrinsic function declarations of `loadByte` and `storeByte` referring to canonical `LOAD_BYTE` and `STORE_BYTE` semantics—both of which may be (re)used by corresponding intrinsics within other types (e.g., an `ObjectReference` type). The benefit of this approach becomes clear as we extend `Address` to include more intrinsic operations, such as the less-than ($<$) intrinsic in Figure 7.5, which is defined in terms of canonical `WORD_LT` semantics, and could again be reused by a number of word-sized types. In the Moxie implementation, individual compilers thus needed only understand how to provide each of the full set of intrinsic semantics once, no matter how many times they were used.

The conspicuous limitation of all the described approaches to providing intrinsic functions is that they require the co-operation of those maintaining the host runtime. This is convenient when the runtime itself is the coding context, but is not a general solution. A more general approach—and the one that we have taken—is to associate the semantics of individual intrinsic operations with `IntrinsicGenerator` instances. These instances understand how to generate the appropriate code for an intrinsic operation. In our current implementation `IntrinsicGenerator` instances are stored in a table indexed by the unique key provided at the intrinsic function declaration (e.g., `"LOAD_BYTE"` in Figure 7.5). Currently, our `IntrinsicGenerator` instances must be

implemented with knowledge of the compiler internals (to allow the intrinsics to code their own semantics), but in the future we intend to allow intrinsics to be constructed more generally, through either providing a set of compiler-neutral building blocks, or the use of a specialized language such as CISL [Moss et al., 2005].

7.2.2.2 Semantic Regimes

Recall that Section 7.1.2.2 introduced the idea of statically scoped semantic regimes which change the default language semantics. When the compiler encounters code that is marked with a semantic regime it treats it specially. Currently, support for individual semantic regimes must be hard-coded into the compiler. This includes turning on and off language features such as bounds-checks, the use of locking primitives, and the presence of yield points. It also includes allowing (or disallowing) calls to certain language features, such as calls to `new()`, or the use of unchecked memory operations (e.g., `@UncheckedMemoryAccess` in Figure 7.1). This mechanism is essential to our objective of containment, allowing the finely specified, well scoped declaration of a region with changed semantics.

7.3 Deployment

The framework described has emerged as the pragmatic consequence of a decade of experience with systems programming in the context of high performance Java Virtual Machines (JVMs), including real-world experience with three Java-in-Java virtual machines [Alpern et al., 1999; Blackburn et al., 2008; Flack et al., 2003], a Java operating system [Prangsma, 2005], and a C/C++ JVM [Apache].

The use of our framework in DRLVM [Apache] (a C/C++ JVM based on the ORP [Cierniak et al., 2005] and StarJIT [Adl-Tabatabai et al., 2003] code bases) is particularly interesting. DRLVM uses the framework to express runtime services such as write barriers and allocation sequences in Java. Our Java-based framework made the code easier to express, removed the impedance mismatch between the service code and the user context in which it is called, and allowed the service code to be trivially inlined and optimized into application code. Previously, DRLVM had used ORP's LIL [Glew et al., 2004] to express service code. Aside from providing a more natural medium to express the service code, the use of our framework was motivated by performance [Kuksenko, 2007]. Our framework is used by DRLVM to implement actions including object model operations, class registry access, lock reservation (lock biasing), accessing the current `Thread` object, the object allocation fast path, and garbage collection write barriers.

Jikes RVM makes extensive use of our framework and is the primary environment from which `org.vmmagic` emerged. The memory management subsystem makes particularly heavy use of `org.vmmagic`, principally because it is concerned with accessing raw memory, which is not supported by regular Java semantics. As the single largest user of the framework, we have focused much of the discussion in this section on the memory manager. However, `org.vmmagic` is used throughout Jikes RVM in a

variety of capacities. A few examples of the wide variety of semantic regimes used by Jikes RVM include: stipulating that an object may not move (`NonMovingAllocation`); defining the special semantics of trampoline code, which by definition never returns (`DynamicBridge`); preventing optimization (`NoOptCompile`); asserting callee save semantics for volatiles (`SaveVolatile`); and eliding null checks (`NoNullCheck`). Jikes RVM also makes use of a wide variety of compiler intrinsics, including: atomic operations used to implement locks; memory barrier and cache flushing operations (required when compiling code and initializing classes on architectures with weak memory models); stack introspection (for exception delivery and debugging); and persisting, modifying, and restoring thread state (to support exact garbage collection, green thread scheduling, and exception delivery). The unboxed magic types used by the memory manager (`Word`, `Address`, `ObjectReference`, `Offset`, and `Extent`) are used throughout the JVM.

In the Jikes RVM context, the framework has shown that it is capable of achieving excellent performance [Garner et al., 2007] *and* design characteristics [Blackburn et al., 2004b] in non-trivial systems. These experiences have led us to believe that the approach is broadly applicable.

As new languages emerge [Allen et al., 2008; Chamberlain et al., 2007; Charles et al., 2005; Huang et al., 2008], we hope the designers will carefully consider the possibility of supporting low-level programming, and that they might find our work useful.

7.4 Summary

Hardware and software complexity is making it harder and harder to reason about the environment in which code is written, frustrating the objective of reliable, secure, and maintainable software. This chapter introduced a principled approach to high-level low-level programming, and a concrete framework derived from it. The value of this approach will be underlined in Chapter 8, which discusses real-world experience with the framework.

Chapter 8

High-Performance and Flexibility with MMTk

The `org.vmmagic` framework described in the previous chapter has been shaped through use in several contexts. This chapter describes real-world experience with high-level low-level programming in the Memory Management Toolkit [Blackburn et al., 2004b], a high performance memory management toolkit written in Java that has served as the primary context for the development of `org.vmmagic`. This chapter discusses how MMTk benefits from being written in a high-level language, as well as how the `org.vmmagic` framework has facilitated the development, debugging, and testing of memory management strategies.

This chapter is structured as follows. Section 8.1 discusses why Java was selected as the language for developing MMTk. Section 8.2 then describes some of MMTk's specific low-level programming requirements. The remainder of the chapter is structured around two case studies. The first study, in Section 8.3, is centered around a redesign of a core aspect of MMTk—the transitive closure—that shows how a high-level low-level programming approach can be used to increase flexibility, *without sacrificing performance*. The second example, in Section 8.4, describes the MMTk harness, which takes advantage of the design of the `org.vmmagic` framework to allow MMTk to execute in a virtualized environment, providing powerful debugging, development, and testing facilities.

Sections of this chapter describe work published in the paper "Demystifying Magic: High-level Low-level Programming" [Frampton, Blackburn, Cheng, Garner, Grove, Moss, and Salishev, 2009a].

8.1 Why Java?

MMTk uses Java for two distinct reasons. First, MMTk derives significant software engineering benefits from being implemented in a high-level, strongly typed language [Blackburn et al., 2004b]. Second, MMTk is written in the same language that it was originally designed to support. This avoids an 'impedance mismatch' between application and runtime code, which can provide a significant performance advantages, as shown from the positive experience of Jalapeño [Alpern et al., 2000] and

DRLVM [Kuksenko, 2007]. When the language impedance mismatch is removed, performance critical code (such as object allocation and write barriers) can be inlined and optimized into user code, allowing an optimizing compiler to produce code as good as or better than hand-selected machine code.

The traditional language for implementing memory managers is C, but it would be difficult to use C to build a toolkit as flexible as MMTk. Using C++ may make it possible to achieve an equally flexible structure, but high performance allocation and barriers would require a complex and fragile solution, such as providing hand-crafted IR fragments to the compiler [Glew et al., 2004], or taking DRLVM's approach and using a framework such as `org.vmmagic` for the helper code and C/C++ for the remainder of the memory manager implementation.

The competitive performance of MMTk (running on Jikes RVM) has been demonstrated by several means, including: the inspection of compiled code fragments for performance critical sections; a bottom line performance comparison to production virtual machines; and a direct performance comparison to a high quality C implementation of a memory manager [Garner et al., 2007].

8.2 Low-level Programming Requirements

To illustrate why MMTk requires low-level access to hardware resources, consider the process of tracing an object in a parallel copying garbage collector. Given an object, the collector must:

1. Determine where references to other objects are located in the object, typically by consulting a reference map linked from the object's header; then

2. Take each reference location, load the reference and:

   (a) Determine whether the reference is non-null (if not then we move on to consider the next reference location);

   (b) Determine that the reference does not point to an already copied object (if it does, then we update the reference location to point to the new copy and move on to consider the next reference location),

   (c) Atomically mark the object to ensure only one copy is made (forcing other threads to spin and wait for us to finish if they are also considering it);

   (d) Allocate an area in the target space and copy the contents of the object to it;

   (e) Store a forwarding pointer from the old version of the object to the new copy (also unmarking the object to allow any threads waiting in step 2(c) to continue); and

   (f) Update the reference location to point to the new copy, and then move on to consider the next reference location.

Two things are clear from the above operations. First, many of the operations deal with information that is generally not accessible in a high-level language. Second, given that the number of objects in the heap generally runs into the millions—and that garbage collection can take a significant fraction of execution time (a common rule of thumb is 10%)—any unnecessary overhead (such as that from a foreign function interface) is likely to have an unacceptable performance impact.

8.3 Case Study: An Object Model for Heap Traversal

This section describes experience from a major redesign of MMTk that I led in 2005 with the assistance of Robin Garner. The focus of this discussion is on the redesign of the heap traversal operation.

One of the key stated benefits of using a high-level language is that it allows a more flexible design—a critical concern in the development of a research toolkit such as MMTk. The need to *scan* through the reference fields of an object is almost universal in garbage collection algorithms, forming the basis for the transitive closure required by tracing collectors, and the recursive decrement operation of a reference counter. The redesign discussed in this section made MMTk more flexible in terms of the types of collectors that could be naturally supported, by changing the design of the heap traversal operation.

8.3.1 Original Design

In the original MMTk [Blackburn et al., 2004b], the importance of a transitive closure was reflected by the inclusion of the mechanics to perform such a closure into the base class of all garbage collectors. The base class also managed a parallel work stack, taking objects in turn off the work stack, scanning each for outgoing references, and calling the `traceObject()` method of the concrete garbage collector type for each one. The individual collector could determine what actions needed to be performed, such as marking, copying, and/or pushing the object onto the work stack.

There was one key limitation in the original design: only a single style of transitive closure could be supported in any given configuration. Collection policies that required multiple logical closures over the heap were forced to either include (potentially expensive) branches at the leaf methods, or to duplicate the base functionality supplied by MMTk. Original MMTk generational collectors took the first option, including conditional branches in the leaf methods to detect if a full-heap or nursery collection was in progress, while an initial implementation of a compacting collector took the second, duplicating the MMTk transitive closure functionality to allow two-phase collection [Frampton, 2003].

8.3.2 Solution

During the redesign, we actively sought to take full advantage of the capabilities of high-level languages—particularly object-oriented programming features. In line

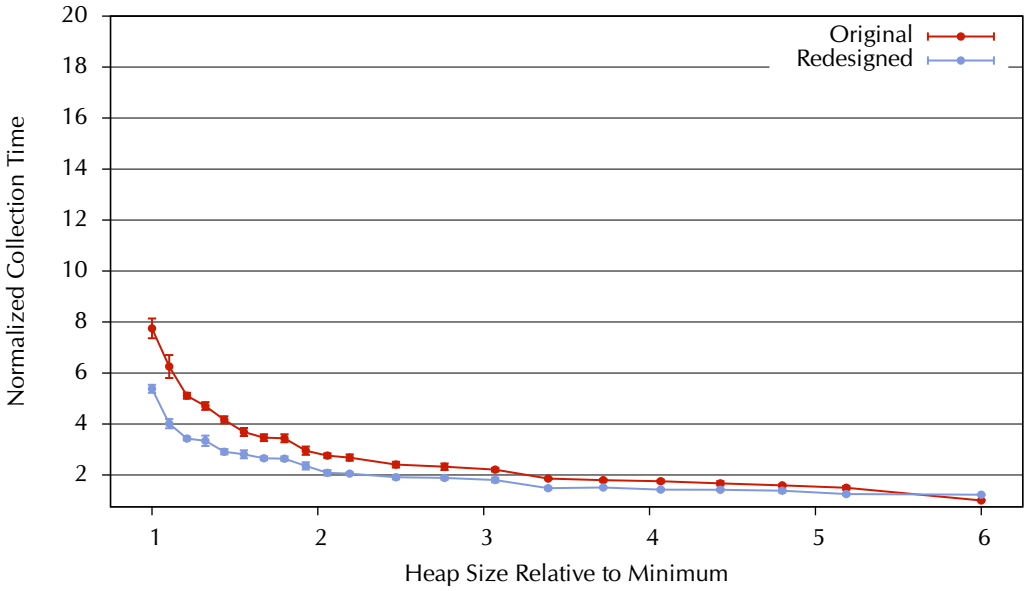


Figure 8.1: Garbage collection performance for the *production* configuration before and after the redesign.

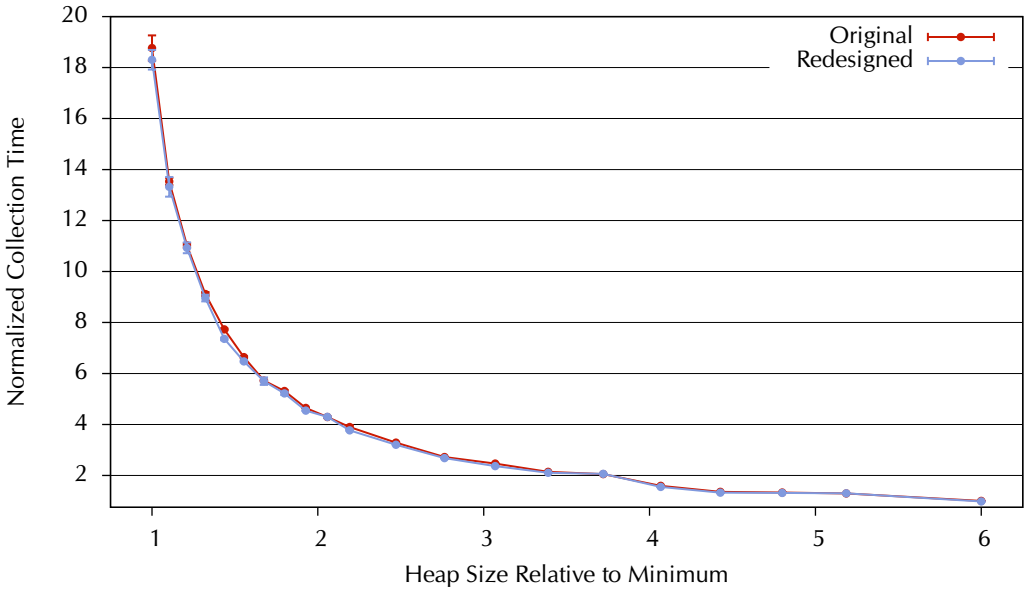


Figure 8.2: Garbage collection performance for the *full-heap mark-sweep* configuration before and after the redesign.

with this goal, the concept of a transitive closure over the heap was brought into a single coherent object model consisting of three key classes:

`TransitiveStep` Declares a single abstract method `processEdge()`, which is called for each reference location found in the object. To trigger these calls, the object to be processed and the `TransitiveStep` to call are passed as a pair to the VM supplied utility method `scanObject()`.

`Trace` Holds the global work stack structures used during a transitive closure over the heap.

`TraceLocal` Manages the thread-local work stack structures, and provides the main processing loop for performing the transitive closure. `TraceLocal` extends the class `TransitiveStep`, implementing `processEdge()` to read the reference value from the location and then call the abstract method `traceObject()`.

This change pushed MMTk toward a more object-oriented programming style, paving the way for research into advanced collection techniques such as Immix [Blackburn and McKinley, 2008], which uses separate `TraceLocal` classes depending on the type of collection. In Immix, this means that the additional checks that are required during a defragmentation collection are performed only when *required*, allowing a faster non-copying `TraceLocal` class to be used for regular collections.

By structuring the tracing operation around well defined interfaces, the design described in this section also makes it easier to implement some classes of optimizations to the tracing loop, such as *specialized scanning*. This involves adding a hidden virtual method to each class, which is a specialized scan method tailored to both the precise layout of references in that class, as well as the appropriate concrete subclass of `TransitiveStep`. These specialized methods are then called in place of the generic `scanObject()` method by the garbage collector.

8.3.3 Performance Evaluation

Figure 8.1 shows the performance of the production configuration of the system immediately before and after the redesign was implemented. Despite the greater reliance on instance methods and a more object-oriented design, the results show that performance was not adversely affected. In fact, garbage collection performance was slightly improved. Recall that the original implementation only allowed each collector to provide a single transitive closure. Because the production configuration of Jikes RVM uses a generational garbage collector, an additional check during tracing is required to determine whether a full-heap or nursery collection is being performed. This check marginally degrades performance, and is not required in the redesigned system. Comparisons of collectors which require only a single type of trace, such as the results for full-heap mark-sweep shown in Figure 8.2, show no performance difference between the original and redesigned systems.

These results have demonstrated that the changes have come at no cost to absolute performance, which is of critical concern to our research. The flexibility improvements are harder in general to quantify, but the implementation of collectors not readily supported by the previous design (e.g., sliding mark-compact and Immix) suggests that the redesign has been a success.

8.4 Case Study: MMTk Harness

While the previous case study showed how MMTk takes advantage of high-level languages to increase flexibility, this section looks at how MMTk harnesses the abstractive power of these languages to help simplify the development of garbage collectors.

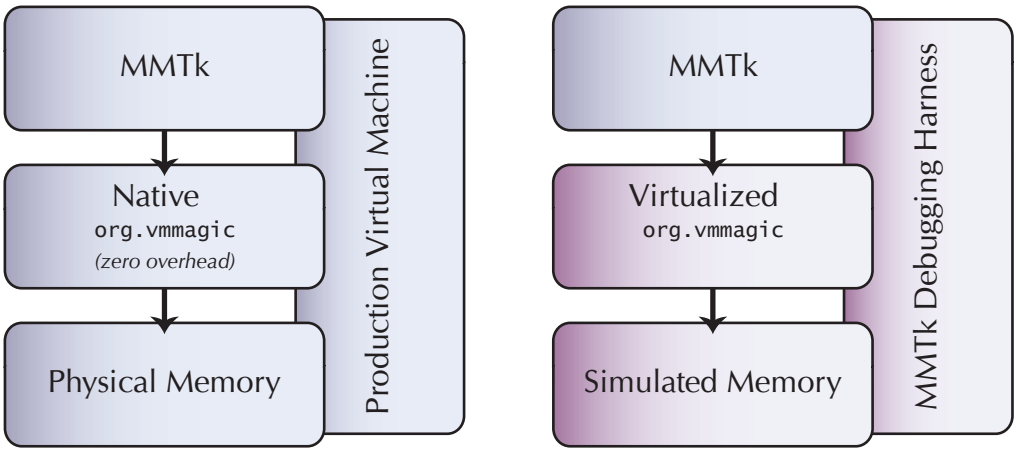
Garbage collectors are notoriously difficult to debug, even when written in a type-safe high-level language. They are very tightly bound to the environment in which they execute: an error in write barrier code could cause pointers to become corrupted, manifesting as errors in user code with no apparent link to the code that produced the error. Modern garbage collectors also tend to be parallel and concurrent, with multiple collector and mutator threads executing at the same time. Modern programming styles also dictate that memory managers support parallel allocation and write barriers, due to the prevalence of multithreaded user code. These characteristics tend to conspire to make debugging a garbage collector within a production JVM a significant challenge.

Our approach is to take the production JVM completely out of the picture, by seamlessly rehosting MMTk in a synthetic, controlled, yet still rich environment: the *MMTk harness*. MMTk can be debugged in this controlled environment using scripts expressed in a simple domain-specific language executed by an interpreter written in Java.

This is possible because all of the low-level functionality required by MMTk is provided by delegating to `org.vmmagic`, whose methods and classes appear as regular Java abstractions. This allows switching between native and virtualized implementations of `org.vmmagic` without requiring changes to MMTk itself.

8.4.1 Harness Architecture

The basic structure of the MMTk harness is illustrated in Figure 8.3, which shows how MMTk is deployed first in a native (production) configuration, and then within the MMTk harness environment. The figure shows that rehosting MMTk onto the interpretive debugging engine requires a) the implementation of a simple virtual machine interface layer targeted at the harness, and b) a *virtualized* implementation of `org.vmmagic` written in pure Java. This figure shows the dependencies of the components, but is not drawn to scale. When running on the MMTk harness, MMTk itself makes up the vast majority of the code complexity. The virtual `org.vmmagic` classes replace the raw memory accesses provided by the virtual machine's compiler intrinsics with a virtualized view of memory, simulated within the harness as a hash table of memory pages (each in turn implemented as an array of integers).



(a) Production MMTk. (b) MMTk harness.

Figure 8.3: MMTk configured to run under a production virtual machine (left) and the MMTk harness (right).

We used a custom domain-specific language for pragmatic reasons. The language syntax is a cut down version of Java syntax, and the language has the minimal set of features required to exercise and validate key memory management functionality. It provides an environment in which it is possible to write simple tests in a clear and concise form. The front-end of the harness is not tied closely to this language, and we have contemplated other front-ends. A Java front-end, for example, could be used to provide access to a wider variety of applications with which to test MMTk collector implementations.

```
1  class Address {
2    int address;
3
4    byte loadByte() {
5      return SimMemory.loadByte(address);
6    }
7
8    void storeByte(byte value) {
9      SimMemory.storeByte(address, value);
10   }
11 }
```

Figure 8.4: Virtualized version of `Address`.

The actual implementation of the virtualized `org.vmmagic` is quite straightforward, with required operations (rather than being executed natively as intrinsics by the host runtime) simply coded as a pure Java implementation within our virtualized memory environment. Figure 8.4 shows an example of how `Address` can be implemented on top of the virtualized memory.

Note that in `org.vmmagic`, magic types such as `Address` and `ObjectReference` are implicitly value types, so need to be passed by value. However, in a pure Java implementation the magic types will be realized as regular Java objects and thus be subject to Java's pass-by-reference semantics. In our current implementation of `org.vmmagic` all magic types are immutable,[1] so pass-by-value and pass-by-reference are semantically equivalent and our pure Java virtualization is trivial. While this is appropriate for the magic types we have required to date, we envisage that requiring immutability will not always appropriate. In such cases, a trivial byte-code rewriting tool could be used to simulate pass-by-value semantics for the appropriate types by inserting copy operations.

8.4.2 Usage Scenarios

The MMTk harness provides many debugging options that are unavailable in existing virtual machine implementations. The following sections describe some of the key scenarios supported by the harness.

8.4.2.1 Unit Testing

One of the primary benefits of the harness environment is that developers can write and execute simple unit tests. This is not possible when running MMTk within Jikes RVM, where both the virtual machine and application allocate and manipulate objects within a single managed heap. Figure 8.5 shows an example test which creates a large volume of cyclic garbage. This test continues to allocate and then discard cycles of garbage. By restricting the heap size and running sufficient iterations, a collector that does not correctly handle cyclic garbage—such as a reference counter without a cycle collector—will run out of memory and crash.

Such unit tests can serve both as development milestones in the construction of a garbage collection algorithm, or as the basis for automated regression testing to ensure the stability of implemented algorithms.

8.4.2.2 Garbage Collector Development

The harness hosts MMTk within a pure Java environment, making it is possible to take advantage of existing debugging tools—such as the Eclipse Java debugger—during the development of garbage collection algorithms. The number, types, and sizes of allocated objects can then be precisely controlled, making it is possible to incrementally develop and test different aspects of a memory management strategy. Programs within the harness can also trigger collection events (such as full heap or nursery collections), as well as intercept key operations performed on objects during collection. This allows simple scenarios to be modeled and then played through the harness to verify that objects are treated as expected by the implementation.

[1] For example, a field `ptr` of type `Address` cannot be incremented. Instead, a reassignment idiom must be used: `ptr = ptr.plus(1);`.

```
1  /**
2   * Create lots of cyclic garbage.
3   */
4  void main() {
5    int cycles = 300;
6    int cycleSize = 100;
7
8    int i = 1;
9    while (i < cycles) {
10     /* Create a garbage cycle by discarding returned value */
11     createCycle(cycleSize);
12     i = i + 1;
13   }
14 }
15
16 /**
17  * Create a cycle of objects of the specified size.
18  */
19 object createCycle(int size) {
20   object head = alloc(1, 10);
21   object tail = head;
22   while(size > 0) {
23     tail.object[0] = alloc(1, 10);
24     tail = tail.object[0];
25     size = size - 1;
26   }
27   tail.object[0] = head;
28   return head;
29 }
```

Figure 8.5: MMTk harness unit test that creates cyclic garbage using the MMTk harness scripting language.

It is also possible to modify the MMTk harness to perform more targeted debugging. This includes placing watch-points at specific addresses within the simulated memory, to either log updates or monitor activity at certain key addresses. This approach is fruitful because the harness provides a controlled environment where scripted operations occur in a deterministic order.

8.5 Summary

This chapter introduced two case studies of high-level low-level programming, both using the `org.vmmagic` framework described in Chapter 7. These studies demonstrate the potential of the high-level low-level programming approach in two key areas. First, in terms of performance, MMTk was competitive with other garbage collection implementations, irrespective of language. Second, in terms of software engineering, the use of a high-level language—and the `org.vmmagic` framework in particular—has made it possible to build more powerful designs and richer tools to assist with testing and debugging. The following chapter continues the theme of

building richer development tools, but shifts focus to discussing the role of visualization in helping to understand complex software systems.

Chapter 9

Visualization with TuningFork

A high-level low-level programming approach provides low-level programmers with only *some* of the tools they need to combat increasing hardware and software complexity. It is also necessary to provide programmers with improved means to understand, debug, and evaluate software. The use of a high-level language may make this need more confronting to some, but it is important to realize that the need exists independent of the use of a high-level programming language.

Applications running on modern runtimes yield a complex multi-layered system. Injecting instrumentation into these layers can be a valuable tool to help us to understand their behavior, but as complexity increases the volume of data produced becomes unmanageable. While gross aggregate statistics (e.g., means, maxima, minima, and order statistics) can be helpful, they are of little value in analyzing fine-grained behavior. The large volumes of information involved suggest taking advantage of the capacity of the human eye to detect patterns and anomalies. This chapter describes *TuningFork*—a visualization tool that I was heavily involved in both designing and developing—that has proven extremely effective in understanding and debugging both real-time applications, and the complex runtime systems upon which these applications execute.

This chapter is structured as follows. First, Section 9.1 gives a brief overview of the environment in which TuningFork was developed. Section 9.2 then discusses how TuningFork relates to other visualization tools. Next, Section 9.3 lists the key requirements for TuningFork, while Section 9.4 shows the architecture TuningFork uses to support these requirements. The *oscilloscope*, a novel visualization technique built using this architecture, is then described in Section 9.5. Last, Section 9.6 details a case study where TuningFork was able to help identify an interesting anomaly in the scheduling behavior of a real-time garbage collector.

This chapter describes work presented in "TuningFork: Visualization, Analysis and Debugging of Complex Real-time Systems" [Bacon, Cheng, Frampton, and Grove, 2007a]. TuningFork was also demonstrated at CC 2006 [Bacon et al., 2006] and OOPSLA 2007 [Bacon et al., 2007b], and is now an open-source project available from http://tuningforkvp.sourceforge.net.

9.1 Introduction

TuningFork is a visualization tool developed as part of the Metronome real-time garbage collection project at IBM Research. Real-time systems are increasing in prevalence and complexity, with automotive, financial, aerospace, telecommunication, and military applications. As the complexity of these systems increases, and the individual cost of developing these systems is driven down, more attention is being focused on the way that the software for such systems is engineered. TuningFork is an Eclipse-based tool that supports the online visualization and analysis of data collated from multiple streams generated in real-time by instrumented subsystems (e.g., applications, virtual machines, and operating systems). TuningFork was the tool used for gathering the data reported on Generational Metronome in Chapter 5.

9.2 Related Work

A large body of work exists on performance visualization and analysis tools.

The concept of dealing with complexity through *vertical profiling*, where multiple layers of complex systems are instrumented and the resulting data correlated, was introduced by Kimelman et al. [1994] with *Program Visualizer*. This approach was extended by Hauswirth et al. [2005] through the use of auto-correlation techniques, rather that relying on user-performed visual correlation.

Parallel and distributed systems are by nature difficult to understand, so it is natural that the use of visualization techniques is prevalent in these communities. Several performance analysis tools have been developed, but the *Pablo* performance analysis environment [Reed et al., 1993] is one of the more complete contributions. Pablo introduces an environment for tracing and analysis, specifies a self-describing trace format (SDDF), and advocates an extensible approach to visualization. Message passing systems have also taken advantage of visualization tools to observe communication activity. *Jumpshot* [Zaki et al., 1999; Wu et al., 2000] is one such system that is targeted at large-scale parallel computations, uses a flexible log file format, and can automatically detect some anomalous behaviors.

Other visualization systems are targeted at application profiling, where the goal is to understand where time is spent during program execution. These tools include *HPCView* [Mellor-Crummey et al., 2002] and *SvPablo* [Rose et al., 1998], which use a combination of hardware performance counters and sampling to hierarchically aggregate the counts and attribute them back to areas of the source code. *Jinsight* [Sevitsky et al., 2001] is a tool designed to assist with the development of Java applications, and consists of a heavily instrumented JVM and a visualization tool. Due to the overheads involved, Jinsight is unsuitable for online usage, although Pauw et al. [2001] show how to restrict it to instrumenting short sections of an execution, allowing the partial analysis of long running programs. *Paradyn* [Miller et al., 1995; Xu et al., 1999; Newhall and Miller, 1999] introduces the ability to dynamically alter what instrumentation is active, in addition to monitoring overheads and adjusting

which instrumentation is active based on acceptable overheads supplied as user parameters.

9.3 Requirements

As discussed earlier, the demands placed on a system for visualizing real-time systems are significant. The more difficult requirements for TuningFork are discussed in this section.

Finding needles in haystacks. There is often a vast amount of information that needs to be generated, processed, and visualized. While for some systems it is appropriate to simply aggregate or summarize information for display, this is not the case for real-time systems in general. Retaining complete raw information can be necessary for debugging or observing some errors; information from a few events in a trace of millions or billions of events may be crucial.

High accuracy timing. Given the tight deadlines that must be met in a real-time system, it is essential that accurate timing information be available when performing analysis. The need to ensure timing accuracy is made more difficult when dealing with multiprocessor systems, where the available timers can be both unsynchronized and have skew relative to each other.

Minimal interference. It is important for the instrumentation not to significantly affect program behavior. In a real-time setting this is a very strict requirement. While it is inevitable that there will be some interference, such interference must be highly predictable, and thus cannot require I/O, allocation, or synchronization. Under load, when the real-time system is having trouble meeting deadlines, it may also be necessary for the logging system to drop events. The rest of the TuningFork system must be robust to this occurrence. Losing events in these circumstances is problematic, but in a live system it is a choice that users may be forced to make.

Live instrumentation. In addition to performing post-mortem offline visualization and analysis, TuningFork is intended to be used *in the field* as an aid to help diagnose and monitor the behavior of live systems. This makes it necessary to be able to connect to such a system and begin observing behavior.

Vertical integration. Because TuningFork is designed to capture and analyze the behavior of complex systems composed of many smaller components and layers, it is necessary to allow the collection and integration of multiple traces. This allows instrumentation of operating systems, runtime systems, applications, libraries, and other devices to be brought together and analyzed within a single system.

Flexibility and extensibility. The requirement for flexibility and extensibility is broad and has several implications. Trace files need to be self-describing, so that traces from many different sources can be easily generated and added. To allow rich visualization of data from different problem domains, it must be possible to integrate custom filters and visualizations. Extensibility allows for the processing and visualization of new types of information, such as creating a musical score from raw midi events, or showing the measured position of a body in space from raw sensor data.

9.4 TuningFork Architecture

In order to meet the requirements set out in the previous section, we developed a software architecture for TuningFork structured around the key components of *traces*, *streams*, and *figures*. Figure 9.1 shows TuningFork's high-level architecture. Each key element of the design is discussed in the following paragraphs.

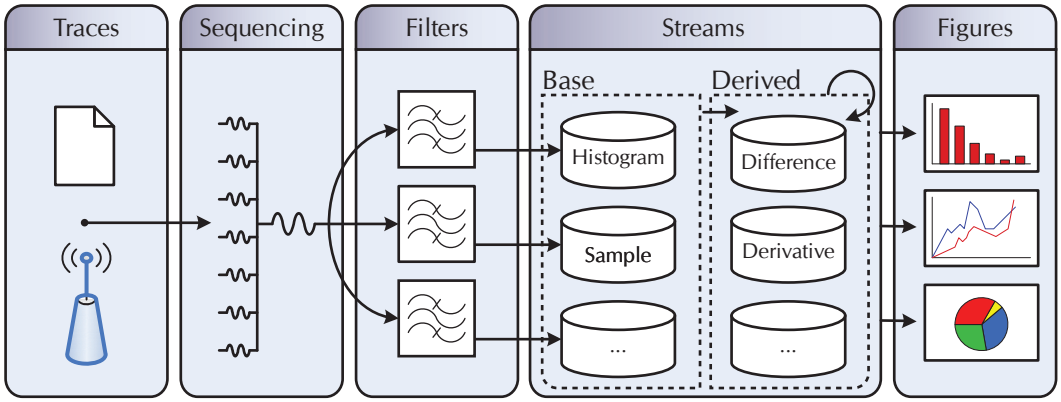


Figure 9.1: The architecture of the TuningFork visualization platform.

Traces. The process of collecting data for visualization begins in an instrumentation layer that generates a *trace*. To date, we have implemented instrumentation layers in Java applications, a real-time Java virtual machine, and the Linux kernel (using SystemTap [SystemTap]). However, any program can generate a trace by complying with a simple binary format. A single TuningFork instance can be simultaneously connected to multiple trace sources, via direct connections over a socket (for online analysis), or via saved binary trace files (for offline analysis). Due to the potentially large size of the underlying trace, TuningFork performs indexing, caching, and summarization of the data stream so that only a manageable portion or summary of the data is held in memory at any time, but it is still possible to analyze all data captured in the trace precisely.

Global event sequencing. A multithreaded application may generate a single trace. To allow this, without requiring significant synchronization and instrumentation

overhead, traces are composed of multiple *feedlets*. Events for each feedlet are time-monotone, and a feedlet is typically produced by a single thread. All feedlets from all traces (remembering that TuningFork can be connected to multiple traces) are subject to a simple merge sort by the *global event sequencer* as the traces are processed by TuningFork.

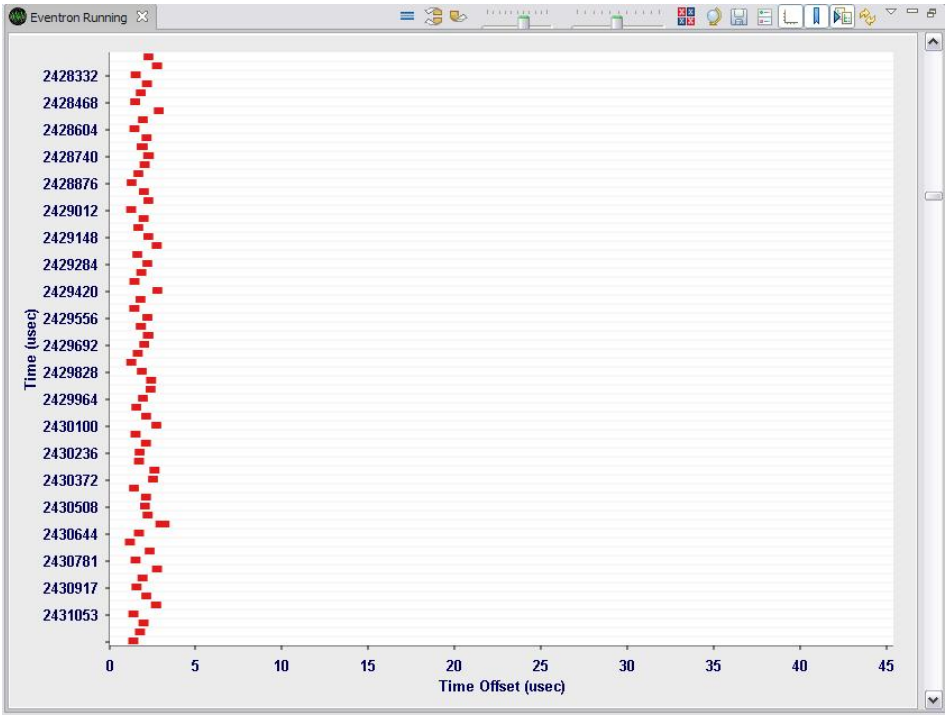
Filters and base streams. For a given TuningFork configuration, a set of filters are run across all events to generate *base streams*. Examples of fundamental base streams include sample streams, which consist of a sequence of (*time, value*) pairs, and interval streams, which consist of a sequence of (possibly overlapping) time intervals as (*start, stop*) pairs.

Derived streams. Basic aggregate statistics, such as minimum, maximum, and average values, are available for all streams. For richer analysis, TuningFork provides a *derived stream* facility, in which new streams can be created based on processing data from other streams. Common uses of this are calculating the rate of change for an underlying stream, filtering out (or in) information in one stream based on intervals in a second stream, or smoothing noisy input functions by averaging their values over larger time windows. Depending on the function being applied, derived streams may create new raw data or be calculated on demand.

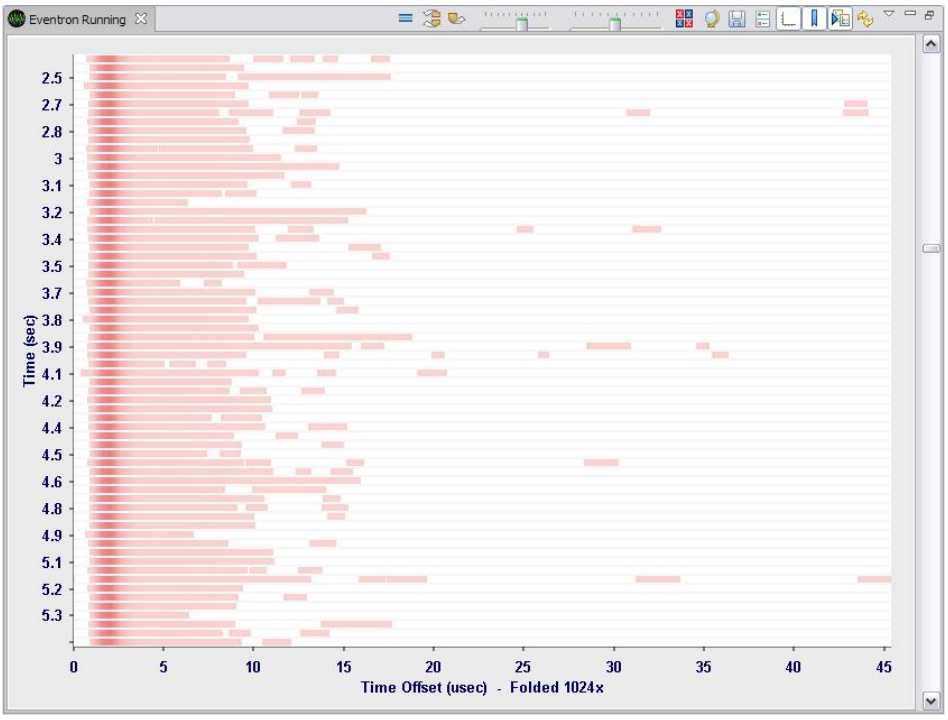
Figures. At the heart of TuningFork is its ability to visualize data by rendering *figures*. TuningFork is capable of rendering figures to various screen and print formats including SWT, PDF, and OpenGL. This is achieved through the TuningFork painting API, which also simplifies the construction of new visualizations by providing appropriate high-level drawing operations. TuningFork provides fundamental visualizations, including time series charts, pie charts, and an *oscilloscope* (discussed below), but is designed to be extensible. Each application area in which we have used TuningFork has made use of this facility, from custom visualizations of the garbage-collected heap in Metronome [Bacon et al., 2003b], to the visualization of MIDI events as a musical score in Harmonicon [Auerbach et al., 2007a].In the design of TuningFork figures, we expended significant effort ensuring they communicate information with efficiency and clarity. This effort has been guided by basic principles on the display of quantitative information—as discussed by Edward R. Tufte in his series of books [Tufte, 1986, 1990, 1997, 2006]—in addition to paying close attention to feedback from users during the development process.

9.5 Oscilloscope Figure

One of the more novel features of TuningFork—described here at length for both its novelty and due to my central role in both design and implementation—is the *oscilloscope* figure, designed to allow the visualization of high-frequency periodic data. The oscilloscope view visualizes time intervals, showing many *strips* of time progressing



(a) Unfolded view showing a total of 2.3ms of data.



(b) Folded view visualizing over 2 seconds of data.

Figure 9.2: Folding in the oscilloscope for a task with a period of 45.3515μs.

from left to right, and from top to bottom. As illustrated through the examples in Figure 9.2, colored blocks represent time intervals in which particular events occur: in this case it is the activity of a periodic real-time task.

The oscilloscope view allows the visualization of large spans of time while still retaining fine detail. For example, using a 1600×1200 pixel display it is possible to precisely visualize events at millisecond resolution while displaying over eight minutes of execution.[1] Even at this resolution, the ability of the human eye to detect patterns makes it quite simple to identify anomalous behavior.

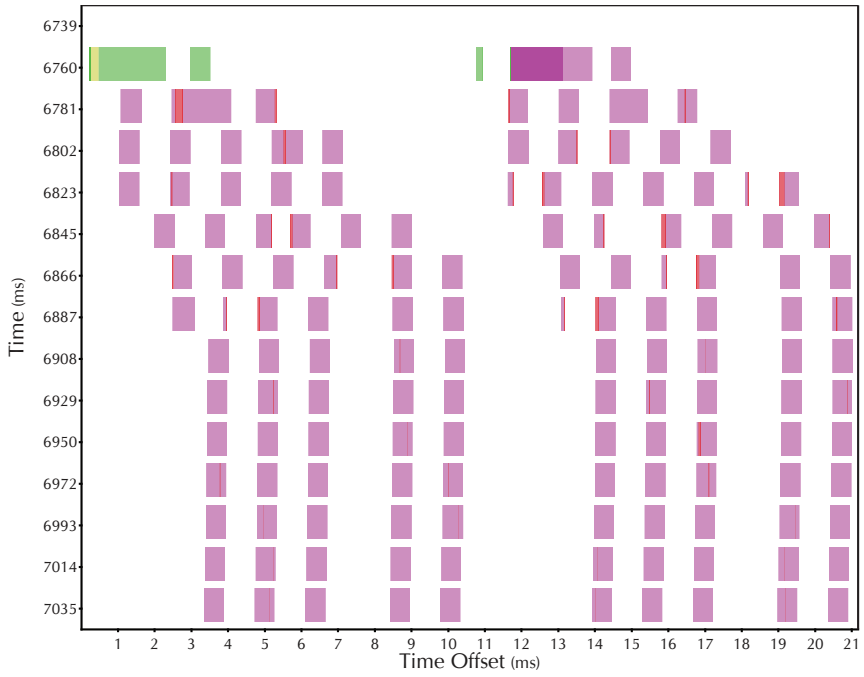
However, there are situations where one needs to view even more data. One application using TuningFork is an audio generator that produces a sound sample at a frequency of 22.05KHz. Each time a sample is generated an event is recorded for the start and end of the operation, resulting in 44,100 events per second (averaging one every 23μs). Visualizing behavior at this timescale requires microsecond resolution, which would allow the precise display of only half of one second at any time when working with the above display.

Folding multiple periods into a single strip makes it possible to visualize an even greater volume of data. When folded, the color intensity at a particular pixel is given by averaging the values at that offset for the periods being displayed in that strip. Thus, periodic behavior shows as dense color, while aperiodic behavior results in a blurred region of lighter color. Naturally this technique is most useful for systems with vast amounts of data and periodic behavior. If the time represented by each strip is set to the natural period of the application, then perfectly scheduled tasks will display as perfectly aligned dark events. Any blurred region shows variance in the scheduling of the tasks. The effect of folding is illustrated by Figure 9.2, which shows a periodic event displayed unfolded (with a single period per strip), and then folded (with 1024 periods per strip). Folding has proven effective at factors up to 1000 and more, making it possible for the display to show minutes of execution while still observing some detail at microsecond resolution. When the natural period of the application is not known in advance, the user can use a slider to vary the display period to discover it. Visual feedback makes it immediately apparent when the correct period has been found, as the image suddenly appears *in focus*, with concentrated regions of color.[2]

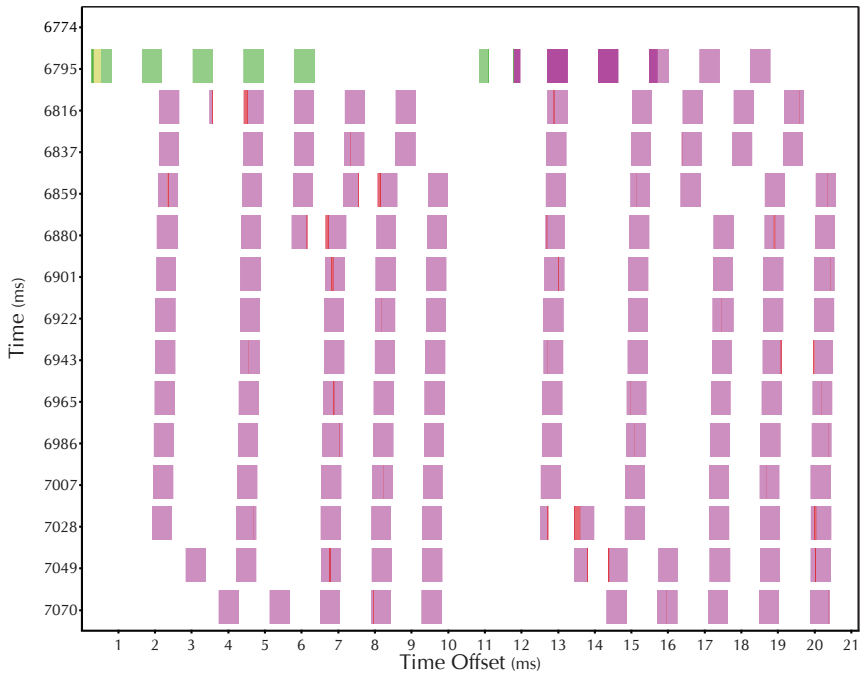
The oscilloscope has proved highly effective at finding interference patterns from other periodic events. For instance, when zoomed out on the view of the audio generation task discussed earlier, we observed a periodic interruption of about 300μs occurring every 50ms. On further investigation this interruption turned out to be the operating system resynchronizing the cycle counter—on which the timing measurements were based—with the lower-frequency crystal oscillator.

[1] This calculation is based on using strips 4 pixels high and a horizontal resolution of 1 pixel per millisecond.

[2] As noted by a reviewer, this view naturally lends itself to autofocus techniques such as analysis in the frequency domain. During the initial development of the oscilloscope we prototyped such a solution, but as the frequency of the events being studied was generally known ahead of time, this feature was not tuned and implemented in later versions.



(a) Original (unexpected) behavior.



(b) Corrected behavior.

Figure 9.3: Oscilloscope view of unexpected and expected scheduling of collector quanta (colors indicate the type of collection activity occurring, which is not relevant to this discussion).

9.6 Case Study: Unexpected Collector Scheduling Decisions

TuningFork was used throughout the development of the Metronome real-time garbage collector for IBM's Real-Time Java product [IBM, 2006]. This section describes an interesting behavioral anomaly in the collector that TuningFork allowed us to diagnose.

Recall from Chapter 2 that Metronome [Bacon et al., 2003b] is an incremental, real-time garbage collector that divides the work of a single garbage collection cycle into a large number of collector quanta. Metronome uses a scheduling algorithm to intersperse mutator execution with collector quanta to achieve a Minimum Mutator Utilization (MMU) [Cheng and Blelloch, 2001] target. As an example, the default settings for the production version of Metronome specify collector quanta of 500μs, and a target MMU of 70% within each 10ms window. This guarantees (when the system is working as designed) that in any 10ms window of execution there will be at most 6 collector quanta, each of 500μs duration.

Visualizing Metronome collector scheduling behavior with TuningFork led to an unexpected discovery of unusually long mutator pauses. This was due to a suboptimal scheduler implementation within Metronome.

First, some background on the Metronome scheduler, which is responsible for controlling mutator and collector interleavings to ensure that both the MMU requirement is satisfied, and that sufficient collector activity is undertaken. In satisfying the MMU bound, the scheduler has some flexibility in how to interleave collector and mutator activity. To ensure that MMU is not violated, the scheduler dynamically computes the current MMU based on a trailing window of previous scheduling decisions. Using the default parameters described above (500μs collector quanta and an MMU target of 70% measured over a 10ms window), it is safe for the scheduler to decide to schedule a collector quantum any time that the current dynamic MMU is at least 75%. When a collection is in progress, the scheduler is invoked every 500μs to make this decision.

The initial implementation of the scheduler strictly followed a simple MMU-centric policy, greedily scheduling collector quanta whenever doing so would not violate the MMU specification. This resulted in the irregular scheduling pattern shown in Figure 9.3(a). At the start of the collection cycle, dynamic MMU was 100%, so the scheduler would initiate several consecutive collector quanta. This clumping quickly dampened as collection proceeded, but it resulted in an irregular schedule with perceived application pauses of up to 3ms. This scheduling algorithm was in use for over a year before visualizing it in TuningFork made the undesirable (but technically correct) behavior immediately obvious. We then revised the scheduling algorithm to avoid this behavior, resulting in the more predictable scheduling behavior shown in Figure 9.3(b).

This is an example of where visualization enables the observation of unexpected or unintuitive behavior, allowing the resolution of issues earlier in the development process. While it may be possible to develop non-visual tools to inspect traces to discover *known* issues, it is difficult to match the ability of an expert user to take

visual information and identify anomalous conditions by relating *observed* behavior to their intuitive models of *expected* behavior.

9.7 Summary

Previous chapters focused on techniques to allow low-level programs to be both expressed and executed within a high-level language environment. This chapter highlighted the important role that visualization tools can play in helping us to better understand the software that executes in this increasingly complex environment. In our experience with TuningFork and Metronome, visualization proved to be of enormous value in understanding and evaluating complex real-time systems. We believe the utility of visualization techniques is only set to increase, given the current trajectory of hardware and software complexity.

Chapter 10

Conclusion

The benefits of modern managed languages—and the type- and memory-safety properties they provide—are evidenced by the position they hold as the standard for implementation in a wide range of application areas. Their use is still far from universal, however, and critical application areas, such as real-time and systems software have been slow to move to these languages, despite being desperate to make improvements in security and reliability.

This thesis seeks to bring the advantages of high-level languages to low-level programming, showing both *why* high-level languages will benefit low-level programming, and demonstrating *how* we can write and run high-level low-level software.

To understand *why* we must turn to high-level languages, this thesis relates our current environment to that of the 1970s, a period when changes in the demands placed on software—alongside increases in hardware complexity—led to a significant shift in the dominant language for programming low-level applications. Looking forward, changes in hardware complexity appear to be accelerating, with trends towards increased levels of concurrency, greater heterogeneity, and complex non-uniform memory hierarchies conspiring to make developing low-level programs increasingly difficult. This raises the question of whether it is again time to look towards higher levels of abstraction to assist with the development of these programs.

Experience we have gained through the continued development of MMTk serves to underscore the value high-level programming languages can bring to low-level programming. Taking advantage of high-level language features, while writing code that expresses low-level concerns, can greatly simplify the process of developing low-level systems. That the resultant system can deliver equivalent, or improved, performance is a compelling justification of the high-level low-level programming approach.

To address *how* to write high-level low-level programs, this thesis identifies three key technical issues holding back the use of high-level languages for low-level programs: 1) the inability to *express* low-level concerns, 2) *garbage collectors* that do not meet key performance requirements, and 3) inadequate *development tools* hindering the low-level programmer in attempts to truly understand program behavior. Each of these concerns have been addressed by work in this dissertation.

We addressed the need to express low-level ideas in high-level languages through the development of the `org.vmmagic` framework. This framework allows the seam-

less integration of low-level *abstractions* into a high-level language, allowing a more controlled and gradual progression between high-level and low-level programming concerns.

Two novel garbage collection algorithms, *Cycle Tracing* and *Generational Metronome*, demonstrate that it is possible to develop garbage collection algorithms that can satisfy the stringent requirements of low-level programs. The two algorithms take complementary approaches. Cycle Tracing is based on a state-of-the-art generational reference counting collector, improving its behavior for programs that exhibit cyclic garbage. Generational Metronome is based on a real-time garbage collector, attaining significant throughput and memory consumption improvements by bringing incremental generational collection to a real-time collector, while retaining the original system's real-time guarantees.

It is clear from trends in hardware and software complexity that the problem of truly *understanding* the behavior of low-level systems is one that is becoming increasingly difficult. Experience with TuningFork has demonstrated the power of visualization techniques, as it makes it possible to interrogate and reason across vast volumes of data, such as is required to observe the behavior of each layer of the software stack.

In combination, these contributions demonstrate that it is both *possible* and *beneficial* to use high-level languages to build high-performance low-level applications.

10.1 Future Work

Each of the lines of investigation described in this thesis has a future—from extending the effective prefetch mechanism to copying garbage collectors; to implementing Cycle Tracing in a fully concurrent reference counting environment; to building on the extensible TuningFork framework to bring richer visualizations to a whole range of potential applications. The following two sections, however, focus on directions that may have the most significant impact in shifting us toward general acceptance and use of the high-level low-level programming approach.

10.1.1 Garbage Collection for Low-level Programs

Generational Metronome provides significant throughput and memory usage improvements over previous real-time garbage collectors. The increased throughput is partly due to improved mutator performance from the use of bump pointer allocation. Immix [Blackburn and McKinley, 2008] has demonstrated the potential of the mark-region approach, which combines fast bump pointer allocation in a mostly non-moving collector. It may be challenging to develop an accurate model of Immix for real-time collection, but the core ideas seem a natural fit for a concurrent or incremental setting.

Avoiding copying has additional advantages in a real-time setting, making the exploration of non-copying generational approaches a natural extension of Generational Metronome. Applying techniques similar to the *sticky mark bit* collection

technique of Demers et al. [1990] could dramatically improve the collection rate of nursery collections. This would have implications beyond simply reducing collection time, because, as our model shows, collection rate dictates the amount of memory required to hold the smallest possible nursery for a given application. A non-copying nursery would also make it more feasible to transition to concurrent (rather than incremental) collection; the additional cost of performing concurrent (as opposed to incremental) copying collection would be necessary only when defragmenting the mature space.

10.1.2 High-level Low-level Programming

The approach to high-level low-level programming outlined in this thesis has evolved from a decade of real-world experience, and has been proven in practice through several projects. However, there is significant promise for future work in a number of directions.

Currently, the definition of semantic regimes and intrinsics in our framework is essentially restricted to developers who are able to modify the language runtime. We have considered two alternatives to address this shortcoming. First, it may be beneficial to design a *standard set* of semantic regimes and intrinsics; this would avoid individual users reinventing the wheel, as well as provide a basic set of functionality to make the framework more approachable. The process of developing such a standard set could also allow additional thought to be put into how the various components of the set may interact. While extensibility would remain an essential component of our approach, we feel that providing such a set may make the framework a more useful starting point for other projects. Such an effort could be along the lines of the `org.vmmagic.unboxed.Address` family of classes, which are used across several projects. A second alternative for opening up the framework would be providing the tools to allow users to specify language extensions, in the form of intrinsics and possibly semantic regimes, in a form independent of the internals of a particular virtual machine implementation.

Bibliography

Abrahams, P. W.; Barnett, J. A.; Book, E.; Firth, D.; Kameny, S. L.; Weissman, C.; Hawkinson, L.; Levin, M. I.; and Saunders, R. A., 1966. The LISP 2 programming language and system. In AFIPS '66 (Fall): *Proceedings of the November 7–10, 1966, Fall Joint Computer Conference* (San Francisco, California, USA, Nov. 1966), 661–676. ACM, New York, New York, USA. doi:10.1145/1464291.1464362. (cited on page 9)

Adl-Tabatabai, A.-R.; Bharadwaj, J.; Chen, D.-Y.; Ghuloum, A.; Menon, V.; Murphy, B.; Serrano, M.; and Shpeisman, T., 2003. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7, 1 (Feb. 2003), 19–31. (cited on page 102)

Agarwal, V.; Hrishikesh, M. S.; Keckler, S. W.; and Burger, D., 2000. Clock rate versus IPC: The end of the road for conventional microarchitectures. In ISCA '00: *Proceedings of the 27th Annual International Symposium on Computer Architecture* (Vancouver, British Columbia, Canada, Jun. 2000), 248–259. ACM, New York, New York, USA. doi:10.1145/339647.339691. (cited on pages 1 and 86)

Aiken, M.; Fähndrich, M.; Hawblitzel, C.; Hunt, G.; and Larus, J., 2006. Deconstructing process isolation. In MSPC '06: *Proceedings of the 2006 Workshop on Memory System Performance and Correctness* (San Jose, California, USA, Oct. 2006), 1–10. ACM, New York, New York, USA. doi:10.1145/1178597.1178599. (cited on page 90)

Allen, E.; Chase, D.; Hallett, J.; Luchangco, V.; Maesse, J.-W.; Ryu, S.; Steele, G. L., Jr.; and Tobin-Hochstadt, S., 2008. *The Fortress Language Specification, Version 1.0*. Sun Microsystems. (cited on page 103)

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* (Denver, Colorado, USA, Nov. 1999), 314–324. ACM, New York, New York, USA. doi:10.1145/320384.320418. (cited on pages 7, 36, 48, 84, 90, 99, 101, and 102)

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 Systems Journal*, 39, 1 (Jan. 2000). (cited on

pages 36, 48, 90, and 105)

Apache. *DRLVM: Dynamic Runtime Layer Virtual Machine*. The Apache Software Foundation. http://harmony.apache.org/subcomponents/drlvm/. Accessed Oct. 2009. (cited on pages 99 and 102)

Appel, A. W.; Ellis, J. R.; and Li, K., 1988. Real-time concurrent collection on stock multiprocessors. In PLDI '88: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA, Jun. 1988), 11–20. ACM, New York, New York, USA. doi:10.1145/53990.53992. (cited on page 20)

Auerbach, J.; Bacon, D. F.; Bömers, F.; and Cheng, P., 2007a. Real-time music synthesis in Java using the Metronome garbage collector. In ICMC 2007: *Proceedings of the 2007 International Computer Music Conference* (Copenhagen, Denmark, Aug. 2007), 103–110. (cited on page 119)

Auerbach, J.; Bacon, D. F.; Cheng, P.; Grove, D.; Biron, B.; Gracie, C.; McCloskey, B.; Micic, A.; and Sciampacone, R., 2008a. Tax-and-spend: Democratic scheduling for real-time garbage collection. In EMSOFT '08: *Proceedings of the 8th ACM & IEEE International Conference on Embedded Software* (Atlanta, Georgia, USA, Oct. 2008), 245–254. ACM, New York, New York, USA. doi:10.1145/1450058.1450092. (cited on page 17)

Auerbach, J.; Bacon, D. F.; Guerraoui, R.; Spring, J. H.; and Vitek, J., 2008b. Flexible task graphs: A unified restricted thread programming model for Java. In LCTES '08: *Proceedings of the 2008 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Tucson, Arizona, USA, Jun. 2008), 1–11. ACM, New York, New York, USA. doi:10.1145/1375657.1375659. (cited on page 17)

Auerbach, J.; Bacon, D. F.; Iercan, D. T.; Kirsch, C. M.; Rajan, V. T.; Roeck, H.; and Trummer, R., 2007b. Java takes flight: Time-portable real-time programming with exotasks. In LCTES '07: *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (San Diego, California, USA, Jun. 2007), 51–62. ACM, New York, New York, USA. doi:10.1145/1254766.1254775. (cited on page 17)

Azatchi, H.; Levanoni, Y.; Paz, H.; and Petrank, E., 2003. An on-the-fly mark and sweep garbage collector based on sliding views. In OOPSLA '03: *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Anaheim, California, USA, Oct. 2003), 269–281. ACM, New York, New York, USA. doi:10.1145/949305.949329. (cited on pages 19 and 62)

Azatchi, H. and Petrank, E., 2003. Integrating generations with advanced reference counting garbage collectors. In CC 2003: *Proceedings of the 12th International Conference on Compiler Construction, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003*, vol. 2622 of *Lecture Notes*

in Computer Science (Warsaw, Poland, Apr. 2003), 185–199. Springer, Berlin/Heidelberg, Germany. doi:10.1007/3-540-36579-6_14. (cited on page 24)

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* (Snowbird, Utah, USA, Jun. 2001), 92–103. ACM, New York, New York, USA. doi:10.1145/378795.378819. (cited on pages 23 and 24)

Bacon, D. F.; Cheng, P.; Frampton, D.; and Grove, D., 2007a. TuningFork: Visualization, analysis and debugging of complex real-time systems. Technical Report RC24162, IBM Research. (cited on page 115)

Bacon, D. F.; Cheng, P.; Frampton, D.; Grove, D.; Hauswirth, M.; and Rajan, V. T., 2006. Demonstration: On-line visualization and analysis of real-time systems with TuningFork. In CC 2006: *Proceedings of the 15th International Conference on Compiler Construction, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006*, vol. 3923 of *Lecture Notes in Computer Science* (Vienna, Austria, Mar. 2006), 96–100. Springer, Berlin/Heidelberg, Germany. doi:10.1007/11688839_8. (cited on page 115)

Bacon, D. F.; Cheng, P.; and Grove, D., 2007b. TuningFork: A platform for visualization and analysis of complex real-time systems. In Companion to OOPSLA '07: *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Montreal, Quebec, Canada, Oct. 2007), 854–855. ACM, New York, New York, USA. doi:10.1145/1297846.1297923. (cited on page 115)

Bacon, D. F.; Cheng, P.; Grove, D.; and Vechev, M. T., 2005. Syncopation: Generational real-time garbage collection in the Metronome. In LCTES '05: *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (San Diego, California, USA, Jun. 2005), 183–192. ACM, New York, New York, USA. doi:10.1145/1065910.1065937. (cited on pages 57, 59, 72, and 73)

Bacon, D. F.; Cheng, P.; and Rajan, V. T., 2003a. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In LCTES '03: *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools For Embedded Systems* (Chicago, Illinois, USA, Jun. 2003), 81–92. ACM, New York, New York, USA. doi:10.1145/780732.780744. (cited on pages 26, 67, and 69)

Bacon, D. F.; Cheng, P.; and Rajan, V. T., 2003b. A real-time garbage collector with low overhead and consistent utilization. In POPL '03: *Proceedings of the 30th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA, Jan. 2003), 285–298. ACM, New York, New York, USA. doi:10.1145/604131.604155. (cited on pages 21, 26, 57, 58, 68, 119, and 123)

Bacon, D. F. and Rajan, V. T., 2001. Concurrent cycle collection in reference counted systems. In ECOOP 2001: *Proceedings of the 15th European Conference on Object-Oriented Programming*, vol. 2072 of *Lecture Notes in Computer Science* (Budapest, Hungary, Jun. 2001), 207–235. Springer, Berlin/Heidelberg, Germany. doi:10.1007/3-540-45337-7_12. (cited on pages 24, 25, 46, and 47)

Baker, H. G., 1992. The Treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27, 3 (Mar. 1992), 66–70. doi:10.1145/130854.130862. (cited on page 20)

Barth, J. M., 1977. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20, 7 (Jul. 1977), 513–518. doi:10.1145/359636.359713. (cited on page 23)

Ben-Yitzhak, O.; Goft, I.; Kolodner, E. K.; Kuiper, K.; and Leikehman, V., 2002. An algorithm for parallel incremental compaction. In ISMM '02: *Proceedings of the 3rd International Symposium on Memory Management* (Berlin, Germany, Jun. 2002), 100–105. ACM, New York, New York, USA. doi:10.1145/512429.512442. (cited on page 21)

Berger, E. D.; McKinley, K. S.; Blumofe, R. D.; and Wilson, P. R., 2000. Hoard: A scalable memory allocator for multithreaded applications. In ASPLOS-IX: *Proceedings of the Ninth International Conference on Architectural Support For Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA, Nov. 2000), 117–128. ACM, New York, New York, USA. doi:10.1145/378993.379232. (cited on page 7)

Bershad, B. N.; Chambers, C.; Eggers, S.; Maeda, C.; McNamee, D.; Pardyak, P.; Savage, S.; and Sirer, E. G., 1994. SPIN: An extensible microkernel for application-specific operating system services. In EW 6: *Proceedings of the 6th Workshop on ACM SIGOPS European Workshop: Matching Operating Systems To Application Needs* (Wadern, Germany, Sep. 1994), 68–71. ACM, New York, New York, USA. (cited on page 90)

Bershad, B. N.; Savage, S.; Pardyak, P.; Sirer, E. G.; Fiuczynski, M. E.; Becker, D.; Chambers, C.; and Eggers, S., 1995. Extensibility, safety and performance in the SPIN operating system. In SOSP '95: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA, Dec. 1995), 267–283. ACM, New York, New York, USA. doi:10.1145/224056.224077. (cited on page 90)

Blackburn, S. M.; Cheng, P.; and McKinley, K. S., 2004a. Myths and realities: The performance impact of garbage collection. In SIGMETRICS/Performance '04: *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems* (New York, New York, USA, Jun. 2004), 25–36. ACM, New York, New York, USA. doi:10.1145/1005686.1005693. (cited on pages 33, 36, and 48)

Blackburn, S. M.; Cheng, P.; and McKinley, K. S., 2004b. Oil and water? High performance garbage collection in Java with MMTk. In ICSE 2004: *Proceedings*

of the 26th International Conference on Software Engineering (Edinburgh, Scotland, UK, May 2004), 137–146. IEEE Computer Society, Los Alamitos, California, USA. doi:10.1109/ICSE.2004.1317436. (cited on pages 36, 84, 103, 105, and 107)

Blackburn, S. M.; Garner, R.; Hoffmann, C.; Khan, 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.; 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* (Portland, Oregon, USA, Oct. 2006), 169–190. ACM, New York, New York, USA. doi:10.1145/1167473.1167488. (cited on pages 32, 37, 48, 50, 74, and 76)

Blackburn, S. M. and Hosking, A. L., 2004. Barriers: Friend or foe? In ISMM '04: *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, British Columbia, Canada, Oct. 2004), 143–151. ACM, New York, New York, USA. doi:10.1145/1029873.1029891. (cited on page 14)

Blackburn, S. M. and McKinley, K. S., 2002. In or out? Putting write barriers in their place. In ISMM '02: *Proceedings of the 3rd International Symposium on Memory Management* (Berlin, Germany, Jun. 2002), 175–184. ACM, New York, New York, USA. doi:10.1145/512429.512452. (cited on page 14)

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 Programming, Systems, Languages, and Applications* (Anaheim, California, USA, Oct. 2003), 344–358. ACM, New York, New York, USA. doi:10.1145/949305.949336. (cited on pages 24, 41, 43, and 46)

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* (Tucson, Arizona, USA, Jun. 2008), 22–32. ACM, New York, New York, USA. doi:10.1145/1375581.1375586. (cited on pages 13, 109, and 126)

Blackburn, S. M.; Salishev, S. I.; Danilov, M.; Mokhovikov, O. A.; Nashatyrev, A. A.; Novodvorsky, P. A.; Bogdanov, V. I.; Li, X. F.; and Ushakov, D., 2008. The Moxie JVM experience. Technical Report TR-CS-08-01, Department of Computer Science, Faculty of Engineering and Information Technology, The Australian National University. (cited on pages 90, 99, 101, and 102)

Blanchet, B., 2003. Escape analysis for Java: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25, 6 (Nov. 2003), 713–775. doi:10.1145/320385.320386. (cited on page 34)

Blelloch, G. E. and Cheng, P., 1999. On bounding time and space for multiprocessor garbage collection. In PLDI '99: *Proceedings of the ACM SIGPLAN*

1999 Conference on Programming Language Design and Implementation (Atlanta, Georgia, USA, May 1999), 104–117. ACM, New York, New York, USA. doi: 10.1145/301618.301648. (cited on page 26)

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* (Albuquerque, New Mexico, USA, Jun. 1993), 197–206. ACM, New York, New York, USA. doi:10.1145/155090.155109. (cited on page 88)

Boehm, H.-J., 2000. Reducing garbage collector cache misses. In ISMM '00: *Proceedings of the 2nd International Symposium on Memory Management* (Minneapolis, Minnesota, USA, Oct. 2000), 59–64. ACM, New York, New York, USA. doi:10.1145/362422.362438. (cited on pages 30 and 32)

Boehm, H.-J., 2005. Threads cannot be implemented as a library. In PLDI '05: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, Illinois, USA, Jun. 2005), 261–268. ACM, New York, New York, USA. doi:10.1145/1065010.1065042. (cited on page 88)

Boehm, H.-J. and Weiser, M., 1988. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18, 9 (Sep. 1988), 807–820. doi:10.1002/spe.4380180902. (cited on pages 7 and 88)

Bollella, G. and Gosling, J., 2000. The real-time specification for Java. *IEEE Computer*, 33, 6 (Jun. 2000), 47–54. doi:10.1109/2.846318. (cited on pages 17 and 74)

Borman, S., 2002. Sensible sanitation: Understanding the IBM Java garbage collector, part 1: object allocation. http://www.ibm.com/developerworks/ibm/library/i-garbage1/. Accessed Oct. 2009. (cited on page 13)

Brooks, R. A., 1984. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In LFP '84: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, USA, Aug. 1984), 256–262. ACM, New York, New York, USA. doi:10.1145/800055.802042. (cited on pages 20 and 27)

Burger, D.; Keckler, S. W.; McKinley, K. S.; Dahlin, M.; John, L. K.; Lin, C.; Moore, C. R.; Burrill, J.; McDonald, R. G.; Yoder, W.; and the TRIPS Team, 2004. Scaling to the end of silicon with edge architectures. *IEEE Computer*, 37, 7 (Jul. 2004), 44–55. doi:10.1109/MC.2004.65. (cited on page 86)

Cardelli, L.; Donahue, J.; Glassman, L.; Jordan, I.; Kalsow, B.; and Nelson, G., 1989. Modula-3 report (revised). Technical Report 52, DEC SRC: Digital Equipment Corporation Systems Research Center. (cited on pages 88, 89, 90, and 94)

Chamberlain, B. L.; Callahan, D.; and Zima., H. P., 2007. Parallel programmability and the Chapel language. *International Journal of High Performance Com-*

puting Applications, 21, 3 (Aug. 2007), 291–312. doi:10.1177/1094342007078442. (cited on page 103)

Charles, P.; Grothoff, C.; Saraswat, V.; Donawa, C.; Kielstra, A.; Ebcioglu, K.; von Praun, C.; and Sarkar, V., 2005. X10: An object-oriented approach to non-uniform cluster computing. In OOPSLA '05: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, California, USA, Oct. 2005), 519–538. ACM, New York, New York, USA. doi:10.1145/1094811.1094852. (cited on page 103)

Cheney, C. J., 1970. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13, 11 (Nov. 1970), 677–678. doi:10.1145/362790.362798. (cited on page 11)

Cheng, P. and Blelloch, G. E., 2001. A parallel, real-time garbage collector. In PLDI '01: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA, Jun. 2001), 125–136. ACM, New York, New York, USA. doi:10.1145/378795.378823. (cited on pages 16, 26, 67, and 123)

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* (Boston, Massachusetts, USA, Oct. 2004), 199–210. ACM, New York, New York, USA. doi:10.1145/1024393.1024417. (cited on pages 30 and 32)

Chin, W.; Craciun, F.; Qin, S.; and Rinard, M., 2004. Region inference for object-oriented language. In PLDI '04: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (Washington, D.C., USA, Jun. 2004), 243–354. ACM, New York, New York, USA. doi:10.1145/996841.996871. (cited on page 34)

Choi, J.; Gupta, M.; Serrano, M. J.; Sreedhar, V. C.; and Midkiff, S. P., 2003. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25, 6 (Nov. 2003), 876–910. doi:10.1145/945885.945892. (cited on page 34)

Christopher, T. W., 1984. Reference count garbage collection. *Software: Practice and Experience*, 14, 6 (Jun. 1984), 503–507. doi:10.1002/spe.4380140602. (cited on pages 24 and 25)

Cierniak, M.; Eng, M.; Glew, N.; Lewis, B.; and Stichnoth, J., 2005. The Open Runtime Platform: A flexible high-performance managed runtime environment. *Concurrency and Computation: Practice and Experience*, 17, 5-6 (Apr. 2005), 617–637. doi:10.1002/cpe.v17:5/6. (cited on page 102)

Click, C.; Tene, G.; and Wolf, M., 2005. The pauseless GC algorithm. In VEE '05: *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (Chicago, Illinois, USA, Jun. 2005), 46–56. ACM, New York, New York, USA. doi:10.1145/1064979.1064988. (cited on page 21)

Collins, G. E., 1960. A method for overlapping and erasure of lists. *Communications of the ACM*, 3, 12 (Dec. 1960), 655–657. doi:10.1145/367487.367501. (cited on pages 9 and 22)

Corbató, F. J. and Vyssotsky, V. A., 1966. Introduction and overview of the Multics system. In AFIPS '65 (Fall, Part I): *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I* (Las Vegas, Nevada, USA, Nov. 1966), 185–196. ACM, New York, New York, USA. doi:10.1145/1463891.1463912. (cited on page 85)

Demers, A.; Weiser, M.; Hayes, B.; Boehm, H.; Bobrow, D.; and Shenker, S., 1990. Combining generational and conservative garbage collection: Framework and implementations. In POPL '90: *Proceedings of the 17th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA, Jan. 1990), 261–269. ACM, New York, New York, USA. doi:10.1145/96709.96735. (cited on page 127)

Deutsch, L. P. and Bobrow, D. G., 1976. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19, 9 (Sep. 1976), 522–526. doi:10.1145/360336.360345. (cited on pages 23 and 24)

Dijkstra, E. W.; Lamport, L.; Martin, A. J.; Scholten, C. S.; and Steffens, E. F. M., 1978. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21, 11 (Nov. 1978), 965–975. doi:10.1145/359642.359655. (cited on pages 6, 17, and 19)

Doligez, D. and Gonthier, G., 1994. Portable, unobtrusive garbage collection for multiprocessor systems. In POPL '94: *Proceedings of the 21st ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA, Jan. 1994), 70–83. ACM, New York, New York, USA. doi:10.1145/174675.174673. (cited on pages 19 and 57)

Doligez, D. and Leroy, X., 1993. A concurrent, generational garbage collector for a multithreaded implementation of ML. In POPL '93: *Proceedings of the 20th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA, Jan. 1993), 113–123. ACM, New York, New York, USA. doi:10.1145/158511.158611. (cited on pages 19 and 57)

Domani, T.; Kolodner, E. K.; Lewis, E.; Salant, E. E.; Barabash, K.; Lahan, I.; Levanoni, Y.; Petrank, E.; and Yanorer, I., 2000a. Implementing an on-the-fly garbage collector for Java. In ISMM '00: *Proceedings of the 2nd International Symposium on Memory Management* (Minneapolis, Minnesota, USA, Oct. 2000), 155–166. ACM, New York, New York, USA. doi:10.1145/362422.362484. (cited on pages 19 and 57)

Domani, T.; Kolodner, E. K.; and Petrank, E., 2000b. A generational on-the-fly garbage collector for Java. In PLDI '00: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada, Jun. 2000), 274–284. ACM, New York, New York, USA. doi:10.1145/349299.349336. (cited on pages 19 and 57)

Ecma, 2006a. *C# Language Specification, ECMA-334*. Ecma International. http://www.ecma-international.org/publications/standards/ecma-334.htm. (ISO/IEC 23270:2006). Accessed Oct. 2009. (cited on pages 89, 94, and 96)

Ecma, 2006b. *Common Language Infrastructure (CLI), ECMA-335*. Ecma International. http://www.ecma-international.org/publications/standards/ecma-335.htm. (ISO/IEC 23271:2006). Accessed Oct. 2009. (cited on page 89)

Fähndrich, M.; Aiken, M.; Hawblitzel, C.; Hodson, O.; Hunt, G.; Larus, J. R.; and Levi, S., 2006. Language support for fast and reliable message-based communication in Singularity OS. In EuroSys '06: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (Leuven, Belgium, Sep. 2006), 177–190. ACM, New York, New York, USA. doi:10.1145/1217935.1217953. (cited on page 90)

Fenichel, R. R. and Yochelson, J. C., 1969. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12, 11 (Nov. 1969), 611–612. doi:10.1145/363269.363280. (cited on page 11)

Ferrara, P.; Logozzo, F.; and Fähndrich, M., 2008. Safer unsafe code for .NET. In OOPSLA '08: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Nashville, Tennessee, USA, Oct. 2008), 329–346. ACM, New York, New York, USA. doi:10.1145/1449764.1449791. (cited on page 88)

Fiuczynski, M. E.; Hsieh, W. C.; Sirer, E. G.; Pardyak, P.; and Bershad, B. N., 1997. Low-level systems programming with Modula-3. *Threads: A Modula-3 Newsletter*, 3 (Fall 1997). (cited on pages 90 and 100)

Flack, C.; Hosking, T.; and Vitek, J., 2003. Idioms in OVM. Technical Report CSD-TR-03-017, Purdue University. (cited on pages 90, 99, 100, and 102)

Fletcher, J. G., 1975. No! High level languages should not be used to write systems software. In ACM 75: *Proceedings of the 1975 Annual Conference* (Minneapolis, Minnesota, USA, Oct. 1975), 209–211. ACM, New York, New York, USA. doi:10.1145/800181.810319. (cited on pages 86 and 87)

Fletcher, J. G.; Badger, C. S.; Boer, G. L.; and Marshall, G. G., 1972. On the appropriate language for system programming. *ACM SIGPLAN Notices*, 7, 7 (Jul. 1972), 28–30. doi:10.1145/953360.953361. (cited on page 87)

Frailey, D. J., 1975. Should high level languages be used to write systems software? In ACM 75: *Proceedings of the 1975 Annual Conference* (Minneapolis, Minnesota, USA, Oct. 1975), 205. ACM, New York, New York, USA. doi:10.1145/800181.810317. (cited on page 87)

Frampton, D., 2003. *An Investigation into Automatic Dynamic Memory Management Strategies using Compacting Collection*. Honours thesis, Australian National University. (cited on page 107)

Frampton, D.; Bacon, D. F.; Cheng, P.; and Grove, D., 2007. Generational real-time garbage collection: A three-part invention for young objects. In ECOOP

2007: *Proceedings of the 21st European Conference on Object-Oriented Programming*, vol. 4609 of *Lecture Notes in Computer Science* (Berlin, Germany, Jul. 2007), 101–125. Springer, Berlin/Heidelberg, Germany. doi:10.1007/978-3-540-73589-2_6. (cited on page 57)

Frampton, D.; Blackburn, S. M.; Cheng, P.; Garner, R. J.; Grove, D.; Moss, J. E. B.; and Salishev, S. I., 2009a. Demystifying magic: High-level low-level programming. In VEE '09: *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Washington, D.C., USA, Mar. 2009), 81–90. ACM, New York, New York, USA. doi:10.1145/1508293.1508305. (cited on pages 83, 93, and 105)

Frampton, D.; Blackburn, S. M.; Quinane, L. N.; and Zigman, J., 2009b. Efficient concurrent mark-sweep cycle collection. Technical Report TR-CS-09-02, School of Computer Science, College of Engineering and Computer Science, The Australian National University. (cited on page 41)

Garland, M.; Grand, S. L.; Nickolls, J.; Anderson, J.; Hardwick, J.; Morton, S.; Phillips, E.; Zhang, Y.; and Volkov, V., 2008. Parallel computing experiences with CUDA. *IEEE Micro*, 28, 4 (Jul. 2008), 13–27. doi:10.1109/MM.2008.57. (cited on pages 1 and 86)

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* (Montreal, Quebec, Canada, Oct. 2007), 43–54. ACM, New York, New York, USA. doi:10.1145/1296907.1296915. (cited on pages 29, 84, 97, 103, and 106)

Garthwaite, A. and White, D., 1998. The GC interface in the EVM1. Technical report, Sun Microsystems. (cited on page 7)

Gay, D.; Ennals, R.; and Brewer, E., 2007. Safe manual memory management. In ISMM '07: *Proceedings of the 6th International Symposium on Memory Management* (Montreal, Quebec, Canada, Oct. 2007), 2–14. ACM, New York, New York, USA. doi:10.1145/1296907.1296911. (cited on page 88)

Gay, D. and Steensgaard, B., 2000. Fast escape analysis and stack allocation for object-based programs. In CC 2000: *Proceedings of the 9th International Conference on Compiler Construction, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000*, vol. 1781 of *Lecture Notes in Computer Science* (Berlin, Germany, Mar. 2000), 82–93. Springer, Berlin/Heidelberg, Germany. doi:10.1007/3-540-46423-9_6. (cited on page 34)

Glew, N.; Triantafyllis, S.; Cierniak, M.; Eng, M.; Lewis, B.; and Stichnoth, J., 2004. LIL: An architecture-neutral language for virtual-machine stubs. In VM '04: *Proceedings of the 3rd Virtual Machine Research and Technology Symposium* (San Jose, California, USA, May 2004), 111–125. USENIX, Berkeley, California, USA. (cited on pages 99, 102, and 106)

Gosling, J.; Joy, B.; Steele, G. L., Jr.; and Bracha, G., 2005. *The Java Language*

Specification, Third Edition. Addison-Wesley Professional, 3rd edn. ISBN 0-321-24678-0. (cited on pages 89 and 96)

Graham, R. M., 1970. Use of high level languages for systems programming. Technical report, Massachusetts Institute of Technology. (cited on page 85)

Grossman, D., 2003. Type-safe multithreading in Cyclone. In TLDI '03: *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (New Orleans, Louisiana, USA, Jan. 2003), 13–25. ACM, New York, New York, USA. doi:10.1145/604174.604177. (cited on page 88)

Guyer, S. Z.; McKinley, K. S.; and Frampton, D., 2006. Free-Me: A static analysis for automatic individual object reclamation. In PLDI '06: *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada, Jun. 2006), 364–375. ACM, New York, New York, USA. doi:10.1145/1133981.1134024. (cited on page 29)

Hallgren, T.; Jones, M. P.; Leslie, R.; and Tolmach, A., 2005. A principled approach to operating system construction in Haskell. In ICFP '05: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming* (Tallinn, Estonia, Sep. 2005), 116–128. ACM, New York, New York, USA. doi:10.1145/1086365.1086380. (cited on page 90)

Hauswirth, M.; Diwan, A.; Sweeney, P. F.; and Mozer, M. C., 2005. Automating vertical profiling. In OOPSLA '05: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, California, USA, Oct. 2005), 281–296. ACM, New York, New York, USA. doi:10.1145/1094811.1094834. (cited on page 116)

Henry G. Baker, J., 1978. List processing in real time on a serial computer. *Communications of the ACM*, 21, 4 (Apr. 1978), 280–294. doi:10.1145/359460.359470. (cited on pages 16, 20, and 26)

Herlihy, M. P. and Moss, J. E. B., 1992. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3, 3 (May 1992), 304–311. doi:10.1109/71.139204. (cited on page 21)

Hicks, M.; Morrisett, G.; Grossman, D.; and Jim, T., 2004. Experience with safe manual memory-management in Cyclone. In ISMM '04: *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, British Columbia, Canada, Oct. 2004), 73–84. ACM, New York, New York, USA. doi:10.1145/1029873.1029883. (cited on page 34)

Hirzel, M.; Diwan, A.; and Henkel, J., 2002. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Transactions on Programming Languages and Systems*, 24, 6 (Nov. 2002), 593–624. doi:10.1145/586088.586089. (cited on page 33)

Hirzel, M. and Grimm, R., 2007. Jeannie: Granting Java Native Interface developers their wishes. In OOPSLA '07: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*

(Montreal, Quebec, Canada, Oct. 2007), 19–38. ACM, New York, New York, USA. doi:10.1145/1297027.1297030. (cited on pages 89 and 94)

Horning, J. J., 1975. Yes! High level languages should be used to write systems software. In ACM 75: *Proceedings of the 1975 Annual Conference* (Minneapolis, Minnesota, USA, Oct. 1975), 206–208. ACM, New York, New York, USA. doi:10.1145/800181.810318. (cited on page 87)

Hosking, A. L.; Moss, J. E. B.; and Stefanovic, D., 1992. A comparative performance evaluation of write barrier implementations. In OOPSLA '92: *Proceedings of the 7th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Canada, Oct. 1992), 92–109. ACM, New York, New York, USA. doi:10.1145/141936.141946. (cited on page 14)

Huang, S. S.; Hormati, A.; Bacon, D. F.; and Rabbah, R., 2008. Liquid Metal: Object-oriented programming across the hardware/software boundary. In ECOOP 2008: *Proceedings of the 22nd European Conference on Object-Oriented Programming*, vol. 5142 of *Lecture Notes in Computer Science* (Paphos, Cyprus, Jul. 2008), 76–103. Springer, Berlin/Heidelberg, Germany. doi:10.1007/978-3-540-70592-5_5. (cited on pages 1, 86, 99, and 103)

Hudson, R. L. and Moss, J. E. B., 2001. Sapphire: Copying GC without stopping the world. In JGI '01: *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande* (Palo Alto, California, USA, Jun. 2001), 48–57. ACM, New York, New York, USA. doi:10.1145/376656.376810. (cited on page 22)

Hunt, G.; Larus, J.; Abadi, M.; Aiken, M.; Barham, P.; Fähndrich, M.; Hawblitzel, C.; Hodson, O.; Levi, S.; Murphy, N.; Steensgaard, B.; Tarditi, D.; Wobber, T.; and Zill, B., 2005. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research. (cited on pages 84 and 90)

Hunt, G. C. and Larus, J. R., 2007. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41, 2 (Apr. 2007), 37–49. doi:10.1145/1243418.1243424. (cited on page 90)

IBM, 2006. *WebSphere Real Time Java Virtual Machine*. IBM Corporation. http://www.ibm.com/software/webservers/realtime. Accessed Oct. 2009. (cited on pages 73 and 123)

Inoue, H.; Stefanović, D.; and Forrest, S., 2003. Object lifetime prediction in Java. Technical Report TR-CS-2003-28, University of New Mexico. (cited on pages 33 and 34)

Jim, T.; Morrisett, J. G.; Grossman, D.; Hicks, M. W.; Cheney, J.; and Wang, Y., 2002. Cyclone: A safe dialect of C. In ATEC '02: *Proceedings of the General Track of the 2002 USENIX Annual Technical Conference* (Monterey, California, USA, Jun. 2002), 275–288. USENIX Association, Berkeley, California, USA. (cited on page 88)

Johnson, R. E., 1992. Reducing the latency of a real-time garbage collector. *ACM Letters on Programming Languages and Systems*, 1, 1 (Mar. 1992), 46–58. doi:10.1145/130616.130621. (cited on page 21)

Johnstone, M. S. and Wilson, P. R., 1998. The memory fragmentation problem: Solved? In ISMM '98: *Proceedings of the 1st International Symposium on Memory Management* (Vancouver, British Columbia, Canada, Oct. 1998), 26–36. ACM, New York, New York, USA. doi:10.1145/286860.286864. (cited on page 20)

Joisha, P. G., 2006. Compiler optimizations for nondeferred reference-counting garbage collection. In ISMM '06: *Proceedings of the 5th International Symposium on Memory Management* (Ottawa, Ontario, Canada, Jun. 2006), 150–161. ACM, New York, New York, USA. doi:10.1145/1133956.1133976. (cited on page 23)

Joisha, P. G., 2007. Overlooking roots: A framework for making nondeferred reference-counting garbage collection fast. In ISMM '07: *Proceedings of the 6th International Symposium on Memory Management* (Montreal, Quebec, Canada, Oct. 2007), 141–158. ACM, New York, New York, USA. doi:10.1145/1296907.1296926. (cited on page 23)

Jones, R. E. and Lins, R. D., 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Ltd., New York, New York, USA. ISBN 0-471-94148-4. (cited on pages 5, 7, 18, 24, and 31)

Kahle, J. A.; Day, M. N.; Hofstee, H. P.; Johns, C. R.; Maeurer, T. R.; and Shippy, D., 2005. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49, 4/5 (Jul. 2005), 589–604. (cited on pages 1 and 86)

Kermany, H. and Petrank, E., 2006. The Compressor: Concurrent, incremental, and parallel compaction. In PLDI '06: *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada, Jun. 2006), 354–363. ACM, New York, New York, USA. doi:10.1145/1133981.1134023. (cited on pages 13 and 21)

Kernighan, B. W. and Ritchie, D. M., 1988. *The C Programming Language*. Prentice Hall, Upper Saddle River, New Jersey, USA. ISBN 0-13-110362-8. (cited on pages 85 and 88)

Kimelman, D.; Rosenburg, B.; and Roth, T., 1994. Strata-various: Multi-layer visualization of dynamics in software system behavior. In Visualization '94: *Proceedings of the IEEE Conference on Visualization '94* (Washington, D.C., USA, Oct. 1994), 172–178. IEEE Computer Society, Los Alamitos, California, USA. doi:10.1109/VISUAL.1994.346322. (cited on page 116)

Kuksenko, S., 2007. Suggestion: Let's write some small and hot native(kernel) methods on vmmagics. http://www.mail-archive.com/dev@harmony.apache.org/msg07606.html. Accessed Oct. 2009. (cited on pages 102 and 106)

Landau, C., 1976. On high-level languages for system programming. *ACM SIGPLAN Notices*, 11, 1 (Jan. 1976), 30–31. doi:10.1145/987324.987328. (cited on page 85)

Lea, D. Low-level memory fences. http://gee.cs.oswego.edu/dl/concurrent/dist/docs/java/util/concurrent/atomic/Fences.html. Accessed Oct. 2009. (cited on page 97)

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* (Tampa Bay, Florida, USA, Oct. 2001), 367–380. ACM, New York, New York, USA. doi:10.1145/504282.504309. (cited on pages 23, 43, and 47)

Levanoni, Y. and Petrank, E., 2006. An on-the-fly reference-counting garbage collector for Java. *ACM Transactions on Programming Languages and Systems*, 28, 1 (Jan. 2006), 1–69. doi:10.1145/1111596.1111597. (cited on page 23)

Liang, S., 1999. *The Java Native Interface: Programmer's Guide and Specification*. The Java Series. Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, USA. ISBN 0-201-32577-2. (cited on pages 89, 94, and 95)

Lieberman, H. and Hewitt, C., 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26, 6 (Jun. 1983), 419–429. doi:10.1145/358141.358147. (cited on page 14)

Lins, R. D., 1992. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44, 4 (Dec. 1992), 215–220. doi:10.1016/0020-0190(92)90088-D. (cited on pages 24 and 25)

Lyle, D. M., 1971. A hierarchy of high order languages for systems programming. In *Proceedings of the SIGPLAN Symposium on Languages For System Implementation* (Lafayette, Indiana, USA, Oct. 1971), 73–78. ACM, New York, New York, USA. doi:10.1145/800234.807061. (cited on page 85)

Maessen, J.-W.; Sarkar, V.; and Grove, D., 2001. Program analysis for safety guarantees in a Java virtual machine written in Java. In PASTE '01: *Proceedings of the 2001 ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Snowbird, Utah, USA, Jun. 2001), 62–65. ACM, New York, New York, USA. doi:10.1145/379605.379668. (cited on page 90)

Marinov, D. and O'Callahan, R., 2003. Object equality profiling. In OOPSLA '03: *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Anaheim, California, USA, Oct. 2003), 313–325. ACM, New York, New York, USA. doi:10.1145/949305.949333. (cited on page 34)

Martínez, A. D.; Wachenchauzer, R.; and Lins, R. D., 1990. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34, 1 (Feb. 1990), 31–35. doi:10.1016/0020-0190(90)90226-N. (cited on pages 24, 25, and 46)

McCarthy, J., 1960. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3, 4 (Apr. 1960), 184–195. doi:10.1145/367177.367199. (cited on page 10)

McCloskey, B.; Bacon, D. F.; Cheng, P.; and Grove, D., 2008. Staccato: A parallel and concurrent real-time compacting garbage collector for multiprocessors.

Technical Report RC24504, IBM Research. (cited on page 22)

Mellor-Crummey, J.; Fowler, R. J.; Marin, G.; and Tallent, N., 2002. HPCVIEW: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23, 1 (May 2002), 81–104. doi:10.1023/A:1015789220266. (cited on page 116)

Meyer, M., 2006. A true hardware read barrier. In ISMM '06: *Proceedings of the 5th International Symposium on Memory Management* (Ottawa, Ontario, Canada, Jun. 2006), 3–16. ACM, New York, New York, USA. doi:10.1145/1133956.1133959. (cited on page 21)

Miller, B. P.; Callaghan, M. D.; Cargille, J. M.; Hollingsworth, J. K.; Irvin, R. B.; Karavanic, K. L.; Kunchithapadam, K.; and Newhall, T., 1995. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28, 11 (Nov. 1995), 37–46. doi:10.1109/2.471178. (cited on page 116)

Minsky, M., 1963. A LISP garbage collector algorithm using serial secondary storage. Technical report, Massachusetts Institute of Technology. (cited on page 11)

Moon, D. A., 1984. Garbage collection in a large LISP system. In LFP '84: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, USA, Aug. 1984), 235–246. ACM, New York, New York, USA. doi:10.1145/800055.802040. (cited on page 14)

Moss, J. E. B.; Palmer, T.; Richards, T.; Edward K. Walters, I.; and Weems, C. C., 2005. CISL: A class-based machine description language for co-generation of compilers and simulators. *International Journal of Parallel Programming*, 33, 2 (Jun. 2005), 231–246. doi:10.1007/s10766-005-3587-1. (cited on page 102)

Nethercote, N. and Seward, J., 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In PLDI '07: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA, Jun. 2007), 89–100. ACM, New York, New York, USA. doi:10.1145/1250734.1250746. (cited on page 88)

Nettles, S. and O'Toole, J., 1993. Real-time replication garbage collection. In PLDI '93: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA, Jun. 1993), 217–226. ACM, New York, New York, USA. doi:10.1145/155090.155111. (cited on page 21)

Newhall, T. and Miller, B. P., 1999. Performance measurement of dynamically compiled Java executions. In JAVA '99: *Proceedings of the ACM 1999 Conference on Java Grande* (San Francisco, California, USA, Jun. 1999), 42–50. ACM, New York, New York, USA. doi:10.1145/304065.304093. (cited on page 116)

Oracle. *Oracle JRockit JVM*. Oracle Corporation. http://www.oracle.com/technology/products/jrockit/index.html. Accessed Oct. 2009. (cited on page 13)

Pauw, W. D.; Mitchell, N.; Robillard, M.; Sevitsky, G.; and Srinivasan, H., 2001. Drive-by analysis of running programs. In *Proceedings of the ICSE 2001 Workshop on Software Visualization*, 17–22. (cited on page 116)

Paz, H.; Bacon, D. F.; Kolodner, E. K.; Petrank, E.; and Rajan, V. T., 2007. An efficient on-the-fly cycle collection. *ACM Transactions on Programming Languages and Systems*, 29, 4 (Aug. 2007), 20. doi:10.1145/1255450.1255453. (cited on page 46)

Pizlo, F.; Frampton, D.; Petrank, E.; and Steensgaard, B., 2007. Stopless: A real-time garbage collector for multiprocessors. In ISMM '07: *Proceedings of the 6th International Symposium on Memory Management* (Montreal, Quebec, Canada, Oct. 2007), 159–172. ACM, New York, New York, USA. doi:10.1145/1296907.1296927. (cited on page 22)

Pizlo, F.; Petrank, E.; and Steensgaard, B., 2008. A study of concurrent real-time garbage collectors. In PLDI '08: *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, Arizona, USA, Jun. 2008), 33–44. ACM, New York, New York, USA. doi:10.1145/1375581.1375587. (cited on page 22)

Prangsma, E., 2005. Libre software meeting presentation: Why Java is practical for modern operating systems. http://www.jnode.org/node/681. Accessed Oct. 2009. (cited on pages 90, 99, and 102)

Qian, F. and Hendren, L., 2002. An adaptive, region-based allocator for Java. In ISMM '02: *Proceedings of the 3rd International Symposium on Memory Management* (Berlin, Germany, Jun. 2002), 127–138. ACM, New York, New York, USA. doi:10.1145/512429.512446. (cited on page 34)

Reed, D. A.; Aydt, R. A.; Noe, R. J.; Roth, P. C.; Shields, K. A.; Schwartz, B. W.; and Tavera, L. F., 1993. Scalable performance analysis: The Pablo performance analysis environment. In SPLC '93: *Proceedings of the 1993 Scalable Parallel Libraries Conference* (Starkville, Mississippi, USA, Oct. 1993), 104–113. IEEE Computer Society, Los Alamitos, California, USA. doi:10.1109/SPLC.1993.365577. (cited on page 116)

Richards, M., 1969. BCPL: A tool for compiler writing and system programming. In AFIPS '69 (Spring): *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference* (Boston, Massachusetts, USA, May 1969), 557–566. ACM, New York, New York, USA. doi:10.1145/1476793.1476880. (cited on page 88)

Rigo, A. and Pedroni, S., 2006. PyPy's approach to virtual machine construction. In Companion to OOPSLA '06: *Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Portland, Oregon, USA, Oct. 2006), 944–953. ACM, New York, New York, USA. doi:10.1145/1176617.1176753. (cited on page 90)

Ritchie, D. M., 1993. The development of the C language. In HOPL-II: *Proceedings of the 2nd ACM SIGPLAN Conference on History of Programming Languages* (Cam-

bridge, Massachusetts, USA, Apr. 1993), 201–208. ACM, New York, New York, USA. doi:10.1145/154766.155580. (cited on pages 87 and 88)

Ritchie, S., 1997. Systems programming in Java. *IEEE Micro*, 17, 3 (May 1997), 30–35. doi:10.1109/40.591652. (cited on page 89)

Rose, L. D.; Zhang, Y.; and Reed, D. A., 1998. SvPablo: A multi-language performance analysis system. In Tools '98: *Proceedings of the 10th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools* (Palma de Mallorca, Spain, Sep. 1998), 352–355. Springer, Berlin/Heidelberg, Germany. doi:10.1007/3-540-68061-6_31. (cited on page 116)

Sammet, J. E., 1969. *Programming Languages: History and Fundamentals*. Prentice Hall, Upper Saddle River, New Jersey, USA. ISBN 0-13-729988-5. (cited on page 84)

Sammet, J. E., 1971. Brief survey of languages used for systems implementation. In *Proceedings of the SIGPLAN Symposium on Languages For System Implementation* (Lafayette, Indiana, USA, Oct. 1971), 1–19. ACM, New York, New York, USA. doi:10.1145/800234.807055. (cited on pages 84 and 85)

Sammet, J. E., 1972. Programming languages: History and future. *Communications of the ACM*, 15, 7 (Jul. 1972), 601–610. doi:10.1145/361454.361485. (cited on page 85)

Sansom, P., 1991. Dual-mode garbage collection. In *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, 283–310. Department of Electronics and Computer Science, University of Southampton, Southampton, UK. Technical Report CSTR 91-07. (cited on page 13)

Sevitsky, G.; Pauw, W. D.; and Konuru, R., 2001. An information exploration tool for performance analysis of Java programs. In TOOLS Europe 2001: *Proceedings of the 38th International Conference on the Technology of Object-Oriented Languages and Systems* (Zürich, Switzerland, Mar. 2001), 85–101. IEEE Computer Society, Los Alamitos, California, USA. doi:10.1109/TOOLS.2001.911758. (cited on page 116)

Shaham, R.; Kolodner, E. K.; and Sagiv, M., 2001. Heap profiling for space-efficient Java. In PLDI '01: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA, Jun. 2001), 104–113. ACM, New York, New York, USA. doi:10.1145/378795.378820. (cited on page 33)

Shapiro, J., 2006. Programming language challenges in systems codes: Why systems programmers still use C, and what to do about it. In PLOS '06: *Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support For Modern Operating Systems* (San Jose, California, USA, Oct. 2006), 9. ACM, New York, New York, USA. doi:10.1145/1215995.1216004. (cited on pages 84 and 87)

Siebert, F., 2000. Eliminating external fragmentation in a non-moving garbage collector for Java. In CASES '00: *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis For Embedded Systems* (San Jose, California, USA, Nov. 2000), 9–17. ACM, New York, New York, USA. doi: 10.1145/354880.354883. (cited on page 20)

Simon, D.; Cifuentes, C.; Cleal, D.; Daniels, J.; and White, D., 2006. Java on the bare metal of wireless sensor devices: The Squawk Java virtual machine. In VEE '06: *Proceedings of the 2nd International Conference on Virtual Execution Environments* (Ottawa, Ontario, Canada, Jun. 2006), 78–88. ACM, New York, New York, USA. doi:10.1145/1134760.1134773. (cited on page 90)

SPEC, 1999. *SPECjvm98, Release 1.03*. Standard Performance Evaluation Corporation. http://www.spec.org/jvm98. Accessed Oct. 2009. (cited on pages 32, 37, 50, and 74)

SPEC, 2001. *SPECjbb2000 (Java Business Benchmark), Release 1.01*. Standard Performance Evaluation Corporation. http://www.spec.org/jbb2000. Accessed Oct. 2009. (cited on pages 32, 37, and 50)

Spoonhower, D.; Auerbach, J.; Bacon, D. F.; Cheng, P.; and Grove, D., 2006. Eventrons: A safe programming construct for high-frequency hard real-time applications. In PLDI '06: *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada, Jun. 2006), 283–294. ACM, New York, New York, USA. doi:10.1145/1133981. 1134015. (cited on page 17)

Spring, J. H.; Pizlo, F.; Guerraoui, R.; and Vitek, J., 2007. Reflexes: Abstractions for highly responsive systems. In VEE '07: *Proceedings of the 3rd International Conference on Virtual Execution Environments* (San Diego, California, USA, Jun. 2007), 191–201. ACM, New York, New York, USA. doi:10.1145/1254810. 1254837. (cited on page 17)

Stanchina, S. and Meyer, M., 2007. Mark-sweep or copying? a "best of both worlds" algorithm and a hardware-supported real-time implementation. In ISMM '07: *Proceedings of the 6th International Symposium on Memory Management* (Montreal, Quebec, Canada, Oct. 2007), 173–182. ACM, New York, New York, USA. doi:10.1145/1296907.1296928. (cited on page 21)

Steele, G. L., Jr., 1975. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18, 9 (Sep. 1975), 495–508. doi:10.1145/361002.361005. (cited on pages 17 and 19)

Stepanian, L.; Brown, A. D.; Kielstra, A.; Koblents, G.; and Stoodley, K., 2005. Inlining Java native calls at runtime. In VEE '05: *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (Chicago, Illinois, USA, Jun. 2005), 121–131. ACM, New York, New York, USA. doi:10.1145/1064979.1064997. (cited on pages 89 and 95)

Stroustrup, B., 1986. *The C++ Programming Language*. Addison-Wesley Longman

Publishing Co., Inc., Boston, Massachusetts, USA. ISBN 0-201-12078-X. (cited on pages 84 and 88)

Stroustrup, B., 1993. A history of C++: 1979–1991. In HOPL-II: *Proceedings of the 2nd ACM SIGPLAN Conference on History of Programming Languages* (Cambridge, Massachusetts, USA, Apr. 1993), 271–297. ACM, New York, New York, USA. doi:10.1145/154766.155375. (cited on pages 87 and 88)

Stroustrup, B., 2007. Evolving a language in and for the real world: C++ 1991-2006. In HOPL-III: *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages* (San Diego, California, USA, Jun. 2007), 4–1–4–59. ACM, New York, New York, USA. doi:10.1145/1238844.1238848. (cited on page 87)

Styger, P., 1967. LISP 2 garbage collector specifications. Technical Report TM-3417/500/00 1, System Development Cooperation. (cited on pages 9 and 11)

Sun. *Maxine Research Project*. Sun Microsystems. http://research.sun.com/projects/maxine. Accessed Oct. 2009. (cited on page 90)

SystemTap. *SystemTap*. http://sourceware.org/systemtap/. Accessed Oct. 2009. (cited on page 118)

Tarditi, D.; Puri, S.; and Oglesby, J., 2005. Accelerator: Simplified programming of graphics-processing units for general-purpose uses via data-parallelism. Technical Report MSR-TR-2004-184, Microsoft Research. (cited on page 86)

TIOBE, 2009. *TIOBE Programming Community Index for September 2009*. TIOBE Software. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Accessed Oct. 2009. (cited on page 86)

Titzer, B. L., 2006. Virgil: Objects on the head of a pin. In OOPSLA '06: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Portland, Oregon, USA, Oct. 2006), 191–208. ACM, New York, New York, USA. doi:10.1145/1167473.1167489. (cited on page 17)

Tofte, M. and Talpin, J., 1997. Region-based memory management. *Information and Computation*, 132, 2 (Feb. 1997), 109–176. doi:10.1006/inco.1996.2613. (cited on page 34)

Tridgell, A., 2004. *Using talloc in Samba4*. Samba Team. http://samba.org/ftp/unpacked/talloc/talloc_guide.txt. Accessed Oct. 2009. (cited on page 88)

Tufte, E. R., 1986. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, Connecticut, USA. ISBN 0-9613921-0-X. (cited on page 119)

Tufte, E. R., 1990. *Envisioning Information*. Graphics Press, Cheshire, Connecticut, USA. ISBN 0-9613921-1-8. (cited on page 119)

Tufte, E. R., 1997. *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphics Press, Cheshire, Connecticut, USA. ISBN 0-9613921-2-6. (cited on page 119)

Tufte, E. R., 2006. *Beautiful Evidence*. Graphics Press, Cheshire, Connecticut, USA. ISBN 0-9613921-7-7. (cited on page 119)

Ungar, D., 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In SDE 1: *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pennsylvania, USA, Apr. 1984), 157–167. ACM, New York, New York, USA. doi:10.1145/800020.808261. (cited on pages 14 and 33)

Ungar, D.; Spitz, A.; and Ausch, A., 2005. Constructing a metacircular virtual machine in an exploratory programming environment. In Companion to OOPSLA '05: *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, California, USA, Oct. 2005), 11–20. ACM, New York, New York, USA. doi:10.1145/1094855.1094865. (cited on page 90)

Vechev, M. T. and Bacon, D. F., 2004. Write barrier elision for concurrent garbage collectors. In ISMM '04: *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, British Columbia, Canada, Oct. 2004), 13–24. ACM, New York, New York, USA. doi:10.1145/1029873.1029876. (cited on page 14)

Vechev, M. T.; Bacon, D. F.; Cheng, P.; and Grove, D., 2005. Derivation and evaluation of concurrent collectors. In ECOOP 2005: *Proceedings of the 19th European Conference on Object-Oriented Programming*, vol. 3586 of *Lecture Notes in Computer Science* (Glasgow, Scotland, UK, Jul. 2005), 577–601. Springer, Berlin/Heidelberg, Germany. doi:10.1007/11531142_25. (cited on page 18)

Vechev, M. T.; Yahav, E.; and Bacon, D. F., 2006. Correctness-preserving derivation of concurrent garbage collection algorithms. In PLDI '06: *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada, Jun. 2006), 341–353. ACM, New York, New York, USA. doi:10.1145/1133981.1134022. (cited on page 18)

Venstermans, K.; Eeckhout, L.; and De Bosschere, K., 2006. 64-bit versus 32-bit virtual machines for Java. *Software: Practice and Experience*, 36, 1 (Jan. 2006), 1–26. doi:10.1002/spe.v36:1. (cited on page 96)

Vernooij, J. R., 2008. SAMBA developers guide. http://www.samba.org/samba/docs/Samba-Developers-Guide.pdf. Accessed Oct. 2009. (cited on page 88)

Whaley, J., 2003. Joeq: A virtual machine and compiler infrastructure. In IVME '03: *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators* (San Diego, California, USA, Jun. 2003), 58–66. ACM, New York, New York, USA. doi:10.1145/858570.858577. (cited on page 90)

Whaley, J. and Rinard, M., 1999. Compositional pointer and escape analysis for Java programs. In OOPSLA '99: *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA, Nov. 1999), 187–206. ACM, New York, New York, USA. doi:10.1145/320385.320400. (cited on page 34)

Wilson, P. R., 1992. Uniprocessor garbage collection techniques. In IWMM 92: *Proceedings of the 1992 International Workshop on Memory Management*, vol. 637 of *Lecture Notes in Computer Science* (St. Malo, France, Sep. 1992), 1–42. Springer, Berlin/Heidelberg, Germany. doi:10.1007/BFb0017182. (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 1995 International Workshop on Memory Management*, vol. 986 of *Lecture Notes in Computer Science* (Kinross, Scotland, UK, Sep. 1995), 1–116. Springer, Berlin/Heidelberg, Germany. doi:10.1007/3-540-60368-9_19. (cited on page 7)

Wu, C. E.; Bolmarcich, A.; Snir, M.; Wootton, D.; Parpia, F.; Chan, A.; Lusk, E.; and Gropp, W., 2000. From trace generation to visualization: A performance framework for distributed parallel systems. In SC2000: *Proceedings of SC2000: the ACM/IEEE Conference on High Performance Networking and Computing* (Dallas, Texas, USA, Nov. 2000), 50. IEEE Computer Society, Los Alamitos, California, USA. doi:10.1109/SC.2000.10050. (cited on page 116)

Wulf, W.; Geschke, C.; Wile, D.; and Apperson, J., 1971a. Reflections on a systems programming language. In *Proceedings of the SIGPLAN Symposium on Languages For System Implementation* (Lafayette, Indiana, USA, Oct. 1971), 42–49. ACM, New York, New York, USA. doi:10.1145/800234.807059. (cited on page 85)

Wulf, W. A.; Russell, D. B.; and Habermann, A. N., 1971b. BLISS: A language for systems programming. *Communications of the ACM*, 14, 12 (Dec. 1971), 780–790. doi:10.1145/362919.362936. (cited on pages 85 and 88)

Xu, Z.; Miller, B. P.; and Naim, O., 1999. Dynamic instrumentation of threaded applications. In PPoPP '99: *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Atlanta, Georgia, USA, May 1999), 49–59. ACM, New York, New York, USA. doi:10.1145/301104.301109. (cited on page 116)

Yamauchi, H. and Wolczko, M., 2006. Writing Solaris device drivers in Java. In PLOS '06: *Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support For Modern Operating Systems* (San Jose, California, USA, Oct. 2006), 3. ACM, New York, New York, USA. doi:10.1145/1215995.1215998. (cited on page 90)

Yuasa, T., 1990. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11, 3 (Mar. 1990), 181–198. doi:10.1016/0164-1212(90)90084-Y. (cited on pages 19, 26, and 62)

Zaki, O.; Lusk, E.; Gropp, W.; and Swider, D., 1999. Toward scalable performance visualization with Jumpshot. *International Journal of High Performance Computing Applications*, 13, 3 (Aug. 1999), 277–288. doi:10.1177/109434209901300310. (cited on page 116)