THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-03-02

A Garbage Collection Design and Bakeoff in JMTk: An Efficient Extensible Java Memory Management Toolkit

Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley

September 2003

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical.Reports@cs.anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

`http://cs.anu.edu.au/techreports/`

Recent reports in this series:

A Garbage Collection Design and Bakeoff in JMTk: An Efficient Extensible Java Memory Management Toolkit

Stephen M Blackburn*
Department of Computer Science
Australian National University
Canberra, ACT, 0200, Australia
Steve.Blackburn@cs.anu.edu.au

Perry Cheng
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY, 10598, USA
perryche@us.ibm.com

Kathryn S McKinley
Department of Computer Sciences
University of Texas at Austin
Austin, TX, 78712, USA
mckinley@cs.utexas.edu

ABSTRACT

In this paper, we describe the design, implementation, and evaluation of a new garbage collection framework called the Java Memory Management Toolkit (JMTk). The goals of JMTk are to provide an efficient, composable, extensible, and portable toolkit for quickly building and evaluating new and existing garbage collection algorithms. Our design clearly demarcates the external interface between the collector and the compiler for portability. For extensibility, JMTk provides a selection of allocators, garbage identification, collection, pointer tracking, and other mechanisms that are efficient and that a wide variety garbage collection algorithms can compose and share. For instance, our mark-sweep and reference counting collectors share the free list implementation. We perform a comprehensive and detailed study of collectors including copying, mark-sweep, reference counting, copying generational, and hybrid generational collectors using JMTk in the Jikes RVM on a uniprocessor. We find that the performance of collectors in JMTk is comparable to the highly tuned original Jikes collectors. In a study of full heap and generational collectors, we confirm the significant benefits of generational collectors on a wide variety of heap sizes, and reveal that on very small heaps, collection time is enormous. These experiments add other new insights, such as firmly establishing that for a variety of generational collectors, a variable-size nursery which is allowed to grow to fill all the space not in use by the older generation performs better than a fixed-size nursery. We thus show the utility of extensive collector comparisons and establish the benefits of our design.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

General Terms

Design, Performance, Algorithms

*This work is supported by NSF ITR grant CCR-0085792, and DARPA grant F33615-01-C-1892. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

Submission to OOPSLA-2003, October, 2003, Anaheim, CA.

Keywords

mark-sweep, copying, reference counting, generational, hybrid, Java

1. Introduction

Programmers are embracing high-level languages such as Java and C# which require automatic memory management, i.e., garbage collection. Although, researchers have studied garbage collection for a long time [3, 25, 26, 31, 35, 41], this new reliance on it and the growing effects of locality, has resulted in a number of new collection algorithms and optimizations [11, 12, 16, 34, 40], as well as reconsideration of some old ones [10]. Most collectors are built from scratch without the benefits of shared, extensible components, and as a result, performance comparisons across a range of approaches is problematic and rare [5, 8]. Although a few researchers have developed garbage collector toolkits [16, 27, 28], these toolkits include only copying collectors but not free lists, and are unable to compose disparate policies and mechanisms.

This paper presents the design, implementation, and evaluation of JMTk, a Java Memory Management Toolkit. Our goal was to combine modularity and appropriate abstractions in an infrastructure that made it fast and easy to build efficient new and existing collector algorithms in a variety of settings. To attain portability, JMTk defines a clean interface between the compiler and the collector. (We do not evaluate portability here, but are pursuing a validation.) For efficiency and composability, JMTk implements a variety of shared collector mechanisms and utilities, such as root enumeration and work queues. In addition, collectors define a policy using the appropriate shared underlying space organization for allocation (copying contiguous allocation or free lists), dead object identification, reclamation, and pointer tracking. JMTk composes policies to build hybrid collectors, such as a generational copying nursery and mark-sweep old space.

We use this infrastructure to build a wide variety of copying, mark-sweep, reference counting, copying generational, and hybrid generational collectors in the Jikes RVM, which we describe in detail in Section 4. Although a few experimental studies of garbage collection precede this one [5, 16, 39], ours is unique in its variety of collectors, its thoroughness of heap size exploration, detailed performance measurements, and its apples-to-apples nature.

In the next section, we compare our framework to other frameworks, including explicit memory management toolkits. We then present our framework design, the individual collectors, and some implementation details. Using the SPEC JVM Benchmarks, we compare our implementations of a copying semi-space and a mark-sweep collector to those in the Jikes RVM, since these collectors are the most highly tuned in the Jikes RVM. We include results for garbage collection time, total time, mutator time, and the number of collections. JMTk collectors perform similarly to these collectors.

JMTk makes much better dynamic use of space, but they allocate and collect faster. The former effect is more dominate and thus JMTk out performs the equivalent collectors in the Jikes RVM, but we can still improve our implementations. These results show some of the advantages of our design.

We next compare JMTk's full heap collectors: copying semi-space, mark-sweep, semi-space/mark-sweep (a new full heap hybrid), and reference counting. The mark-sweep collector does better than copying on small heap sizes, but worse on large ones because the memory efficiency tradeoffs vary with memory pressure. All the full heap collectors perform much worse than all of the generational collectors. For small heaps, pure copying generational is worse than a generational hybrid with a copying nursery and mark-sweep old space, but they are indistinguishable in large heaps. Both perform better if they allow the nursery to fill all available free space, rather than using our choice of fixed-size nursery. These results give a taste for the benefits of JMTk as an experimental platform, and show that JMTk has met the design goals.

2. Related Work

This section compares JMTk with several previous garbage collection and memory management toolkits. We also briefly discuss how this paper goes beyond or complements other garbage collection algorithm studies.

2.1 Memory Management Frameworks

The UMass Language Independent GC Toolkit was the first garbage collection toolkit to tease apart the language and collector interface in order to build portable garbage collectors [27]. Systems for Smalltalk, Modula-3, Cecil, and Java [23] used the UMass GC Toolkit. It provided generational copying collectors, and managed memory in fixed-size blocks. It managed each large object directly, using a list associated with each generation. It did not include free lists. Its design was not general enough to include recent copying collectors such as Older-First [40] nor Beltway [16].

These defects led to the design and implementation of GCTk, a more general Garbage Collection Toolkit for Java [16, 18, 39]. This framework provided a single shared implementation of key functions such as scanning and remembered sets. In addition, GCTk implemented copying age-based collectors by separating the collection increment from the heap position [16, 39], but it did not include free-list managers, nor could it mix and match policies. JMTk improves upon GCTk in a number of ways. It creates a cleaner, and we believe portable language/compiler interface. JMTk uses a composable design such that we can easily mix and match policies and mechanisms. It also adds free-list memory managers (e.g., mark-sweep and reference counting), a large object space, the composition of copying and free-list spaces, and dynamic flexible boundaries between different allocation spaces.

A few researchers have also built memory management toolkits for explicit memory management of C/C++ programs [6, 7, 14, 42]. Heap layers is the most general and high performance of these frameworks [14]. It provides multiple and composable heaps. It achieves the performance of existing custom and general-purpose allocators in a flexible, reusable framework. It attains this combination through the use of mixins [20]. Our framework could also probably benefit from mixins that statically express multiple possible class hierarchies, but we have not investigated it here. Explicit memory managers for C/C++ interact with the program only through the malloc and free interface. JMTk has a more complex interface since it also provides functionality such as write barriers and address consistency. C/C++ also limits the manager to free lists, since objects cannot move. The Customizable Memory Management (CMM) framework focuses on automatically collecting these heaps, but uses virtual method calls, thus sacrificing performance [6, 7]. These frameworks thus are not and need not be as general as JMTk.

2.2 Experimental Garbage Collection Evaluation

Many researchers have evaluated a range of memory allocators for C/C++ programs [13, 14, 15, 19, 24, 44]. For example, Attanasio et al. [5], evaluate different parallel collectors on SPECjbb, focusing on the effect of parallelism on throughput and heap size when running at 8 processors. They concluded that mark-sweep and generational mark-sweep with a fixed-size nursery (16 MB or 64 MB) were both equal and the best among all the collectors. Our data shows that the generational collectors are superior but only if one uses a variable-size nursery.

To our knowledge, very few studies have quantitative comparisons of uniprocessor garbage collection algorithms [16, 28, 29, 39], and these studies evaluate various copying and generational collectors. Our results on copying collectors are similar to theirs, but they do not compare with free-list mark-sweep or reference counting collectors. Only a few studies consider the effect of heap size on performance [16, 21, 32, 39], and as we show here, garbage collectors are very sensitive to heap size, and in particular to tight heaps. Hicks et al. [29] and others [17, 30] measure detailed, specific mechanism costs, but do not consider a variety of collection algorithms. Our work thus stands out as the first thorough evaluation of a variety of different garbage collection allocation, collection, and pointer tracking mechanisms. The comprehensiveness of our approach reveals new insights, such as the most space efficient collection algorithms and the performance tradeoffs each strategy makes.

3. Design

This section describes the JMTk design and implementation. It first presents the language and collector interfaces, and the mechanisms for changing from a mutation phase to collection phase. We present our object model and the mechanisms for enumerating roots and pointers. We also discuss how we dynamically manage virtual and physical memory resources. All of these components are shared among all of the JMTk collectors. We then show how we used an object oriented design to make collector components reusable and composable without sacrificing performance.

3.1 Collector Interface

Since one of JMTk's goal is to be portable, the interface between it and the rest of the virtual machine must be as clear and thin as possible without compromising on design flexibility. Here, we present the most important parts of the interface.

3.1.1 Address Types and Unsafe Operations

At first blush, current type theory does not permit us to express a general garbage collector in a generic type-safe language which precludes us from implementing JMTk in pure Java. In JMTk, we choose to extend the language with certain types and functions that, in general, are unsafe. In particular, JMTk expects the runtime system to provide three special types. The first type, `Address`, corresponds to a hardware-specific notion of memory addresses. The second type is `Offset` which expresses the distance between two addresses. Finally, a value of type `Word` corresponds to the value returned by dereferencing an address. Not surprisingly, these types provide methods that allow pointer arithmetic, pointer comparison, memory reads and writes including atomic operations, and casts. It

is through these methods that the collector performs its lowest-level operations, such as allocating and moving objects.

Since Java does not provide extensible primitive types, these special types must be written as Java classes. However, we rely on the underlying VM to treat these types specially. The intended behavior is for values of such types to be *unboxed* such that operations on these types never result in allocation.

3.1.2 Scheduler

In JMTk, the application threads perform memory allocation but dedicated collection threads perform collection. The transfer of control is mediated through an interface between JMTk and the scheduler. When an allocation fails due to memory exhaustion, the JMTk allocation code calls a scheduler method which suspends further operation on that thread and notifies all other threads to stop execution at the next garabage collection *safe-point* in which the application threads are not for example, in the middle of an allocation or calling sequence. When all threads have reached a consistent state, the scheduler wakes up all the collection threads to perform a collection.

3.2 Roots and Object Model

Garbage collection typically begins at the root set which consists of global variables and local variables residing on the threads' stacks. JMTk assumes that the runtime system provides a method for obtaining all roots. In an adaptive system, where the compiler dynamically generates code, code can die before the program exits, and the garabage collector must be able to reclaim it. In general, the interface must cope with the possibility that the collector moves code. Since return addresses are interior pointers into code objects, the root collection interface must handle this special case. In general, the language side must provide an interface to obtain the base object of an interior pointer. JMTk's interface insists that interior references be paired with their base object. The interface also permits parallel root enumeration.

Likewise, JMTk makes few assumptions about the object model but instead relies on an interface to query about object size and the location of pointer fields within an object. Additionally, JMTk can request a space in the object model to store per-object GC state such as forwarding pointers and color bits. JMTk assume that objects are contiguous.

3.2.1 Allocation

To support fast allocation, we use a compiler pragma to force inlining of unsynchronized allocation sequences for the usual case (the sequence is a function of the allocation policy). This requirement coupled with the need to prevent preemption during allocation, requires compliance on the part of the runtime system. Using Java's *throws* mechanism, JMTk notates which methods should not be interrupted by the scheduler. Additionally, a collector may choose to allocate object in different regions depending on a static property of the allocation such as object type, object size, and allocation site. The interface provides all statically known information to the optimizing compiler so it can remove as many unnecessary runtime tests as possible. Finally, JMTk is designed to interact well with an optimizing compiler and expects the compiler to perform constant propagation and to provide a mechanism for indicating exactly when or when not to inline.

3.3 Mechanisms

In this section, we discuss the core components of JMTk. The system is designed to take advantage of object-oriented design with consideration given to performance. For example, we eschew the use virtual function calls in frequent execution paths by using static or final methods.

3.3.1 Utilities

JMTk provides a number of shared helper classes including locks which will synchronize memory, an important property on hardware with a relaxed memory model. JMTk provides special barriers to synchronize collection threads between different phases of collection. Finally, it provides work queues to manage the working set of objects for the collector to trace. Local variants provide fast access to the owning collector thread. In case of queue overflow or underflow, we use a global shared queue to redistribute work. All of these meta-data structures live in the heap.

3.3.2 Memory Resources

JMTk distinguishes between physical and virtual memory resources in order to dynamically choose the right mixture of allocation strategy. Physical resources track the amount of memory that actually contains application data. We divide the physical resources into blocks for convenience and flexibility. For instance, the blocks in the free-list allocator contain only objects of a single size class and their meta-data. In contrast, virtual memory resources track which portion of its virtual memory ranges the allocator has mapped and are in use. A simple example of a virtual memory resource is a *monotone virtual memory resource* that supports contiguous allocation with an operation to logically discard all of its contents. This mechanism implements a copying collector with parallel allocation. The physical resource can exactly match this virtual resource, or it can occupy a fraction of the resource, for example, when only a fraction of the nursery is occupied.

3.3.3 Allocators and Collectors

JMTK distinguishes between local lock-free allocators and globally synchronized allocators. Since there is no restriction on pointers between threads, collection threads must always be synchronized. JMTk supports two types of allocation: bump-pointer allocation and a free-list style allocation typically for non-moving collectors. Corresponding to the allocators are the copying space and a mark-sweep space. Since the free-list allocators support per-object deallocation, this mechanism supports reference counting as well.

3.4 Plans

At the highest level, every memory management system in JMTk provides a plan which exports the ability to allocate, a polling mechanism to determine when to trigger collection, and a reclamation facility. To accomplish these functions, a plan must decide what virtual memory resources it intends to support. At a minimum, it must have a boot image resource for the runtime boot image (*cf.* Section 4.9), an immortal heap to store immortal runtime data, and an area for storing JMTk's internal data. For the application data, the plan must choose one or more additional virtual memory resource and the corresponding virtual memory range. Depending on the algorithm, the collector may specify a write barrier.

Each instance of the plan supports a local allocator which is tied to a corresponding virtual memory resource. The allocation routine then dispatches the request to the appropriate allocator. A plan also specifies an object tracing method for the each virtual memory resource. To dispatch the correct trace method for an object, we considered two methods. In the first, the collector performs an address comparison to determine the virtual memory resource for the object. The second uses JMTk's global virtual memory resource table to look up the object's virtual memory resource. The

second method is more scalable with respect to the number of virtual memory resources and is more amenable to discontinuous or adaptive spaces. Since this dispatch requires an additional memory load, there might be a performance concern. To determine the cost of this overhead, we measured both variants using the semi-space collector, which has the fewest memory resources, on a benchmark with small average object size (4 fields). The slowdown for table look up is very small; it slows collection throughput by about 0.8%.

3.4.1 Using Inheritance

The many similarities between different garbage collectors is reflected in the use of inheritance in JMTk. For example, the base virtual memory resource class supports the acquisition and release of arbitrary memory pages. The monotone subclass imposes contiguous allocation behavior with a single discard operation. By eliminating the discard operation, we derive an *immortal* heap. Finally, the boot image is a special case of the immortal heap where no further allocation is permitted. In a different hierarchy, the base plan contains the three mandatory spaces and support for pointer fields, interior pointers, work management, and command-line options. Subclassed from the abstract base class is another abstract class (stop-the-world) which provides support for starting and stopping a collection, root computation, and work queue management. One of the subclasses of the stop-the-world class is an abstract generational plan which adds a nursery space, an optional large-object space, and a write barrier. Finally, one particular concrete subclass is the generational hybrid mark-sweep collector discussed in the following section.

4. Collectors

Most of the collectors we use are well established in the literature [31]. We first give a short description of the collectors that we implement in JMTk. We then categorize them, discuss each in detail, and include some additional shared implementation details.

SS: The semi-space collector uses one policy on the whole heap: bump-pointer allocation with a collector that traces and copies live objects.

FG: The fixed-size nursery generational collector uses the SS policy in both a fixed-size nursery and the mature space.

VG: The variable-size nursery generational collector [3] is the same as FG, except it uses all available memory that is not consumed by the mature space for the nursery.

BW: The Beltway collectors add incrementality and older-first principles to generational copying collection.

MS: The mark-sweep collector uses one policy on the whole heap: a free-list allocator and a collector that traces and marks live objects, and then reclaims unmarked objects.

FG-MS: The generational mark-sweep hybrid uses a SS policy in a fixed-size nursery, and a MS policy in the mature space.

VG-MS: This collector is the same as FG-MS, except that it has a nursery that dynamically adjusts to use all available memory.

VC-MS: This copy mark-sweep hybrid collector is a whole heap collector. It uses a copying variable-size young space and MS free list old space. On every collection, it promotes nursery survivors to the MS space, and traces the MS space. This collector requires no write barrier.

RC: The deferred reference-counting collector uses a free-list allocator and a collector that periodically processes mutator increments and decrements for heap objects, deleting objects with a reference count of zero.

We categorize these collectors as follows. FG, VG, FG-MS, and VG-MS are the *generational* collectors which divide the heap into a nursery and an old generation. The *whole heap* collectors, SS, MS, VC-MS, and RC, scavenge the entire heap on every collection. SS, MS, and RC use only one policy. Pure *copying* collectors, SS, FG, and VG, only ever copy objects, and the pure *non-copying* collectors are RC and MS. *Hybrid* collectors use multiple policies. Here we use copying on the young space and a free-list old space for FG-MS, VG-MS, and VC-MS.

For each collector, we now describe the collection triggering mechanisms, the write barrier, space overhead, and time overhead. We fix the heap size in our implementation to ensure fair comparisons.

4.1 SS: Semi-Space

A semi-space collector [22] divides the heap into two halves, a *to-space* and *from-space*, reserving half for copying into (since in the worst case all objects could survive) and half for allocation. It allocates using a bump pointer into the to-space until it is full. It then swaps to-space and from-space, scans all of the reachable objects in from-space, copies them to new to-space, and begins allocating into to-space again. It uses a breadth-first order to traverse all reachable objects. It does not have a write barrier. Collection time is proportional to the number of survivors. Its throughput performance suffers because it repeatedly copies objects that survive for a long time, and its responsiveness suffers because it collects the entire heap every time.

All of the copying collectors (SS, FG, VG, FG-MS, VG-MS, VC-MS) use the same allocation and collection mechanisms (and code!) for the young objects (nursery, to-space) with the same resultant space and time overheads.

4.2 FG: Fixed-Size Nursery Generational

The fixed-size nursery two-generation collector uses the SS policy in both the nursery and old space [35, 41]. Filling the nursery of size N triggers a collection that copies the survivors to the old generation. Filling the entire usable heap triggers a whole heap collection in which FG copies survivors back into the old generation. To collect the nursery independently of the higher generation, FG tracks pointers from the older generation into the nursery. We use a standard generational write barrier [17] that tests whether new pointers cross the nursery address boundary in the old-to-young direction (the nursery resides in high memory). Because object lifetimes typically follow the weak-generational hypothesis, it exhibits good throughput. Its average pause time is also good because it is proportional to the nursery survivors, but its worst case pause time is proportional to collecting the entire heap.

4.3 VG: Variable-Size Nursery Generational

The variable-nursery generational copying collector [3, 16] makes efficient use of memory by allowing the nursery to dynamically use all available memory. In particular, the nursery can grow to half of the heap size less the current size of the mature space (half the heap must always be kept in reserve for copying). When the nursery hits this limit, it triggers a nursery (minor) collection. When the older generation is full, it collects the entire heap. If the nursery size drops below some small fixed threshold, a major collection is also triggered in order to prevent arbitrarily inefficient minor collections. Its write barrier records old-to-nursery pointers by testing

if the new pointer crosses the nursery address boundary in the old-to-young direction. As we show in the results section, its throughput is even better than FG because it utilizes the heap more fully and has fewer collections, but its average pause times suffer because the nursery is larger on average. Maximum pause times are similar to FG.

4.4 BW: Beltway

Beltway collectors are generational copying collectors that divide the heap into belts of FIFO increments [16]. Belts are analogous to generations, but more general since they are divided into fixed size increments that are collected independently. A Beltway configuration X.Y.100 uses a fixed-size nursery which is at most X% of the heap on the first belt. The second belt consists of multiple increments, each at most Y% of the total heap, and the last belt can grow to consume all of the heap. When the heap is full, the collector selects the youngest belt with a full increment, and collects it. The collector promotes to the end of the next higher belt, and collects from the front. Beltway configurations use a dynamically sized copy reserve based on the worst case survival rate of the next collection (accounting for the potential cascading of collections). At the worst, the copy reserve is half the heap, but it is typically much less. Its write barrier is more expensive than a generational collector since it needs to track pointers between increments as well as belts. We are only just getting the Beltway framework working in JMTk, and will report its results in the final paper.

4.5 MS: Mark-Sweep

The mark-sweep collector organizes the heap space using a free list. Our MS implementation partitions blocks of memory into size classes [33, 43], and allocates an object into an available slot in the block of the smallest size class in which it fits. MS allocates a new block of the requested size class if no slot is available. It recycles blocks to different size classes only if the block becomes completely free. It triggers collection when the heap is full. For each block, MS keeps a linked list of free slots, an *inuse* bit map and a *mark* bit map. During a collection, it scans the reachable objects, and marks them as reachable by setting the corresponding bit in the mark bit map. For each block, it identifies dead objects by an exclusive-or of the mark and inuse bit maps, and links each free object onto the free list.

Tracing is proportional to the number of live objects, and reclamation is proportional to the number of free objects. Thus, collection is proportional to the total number of objects in the space. The space requirements include the live objects, bit maps, and fragmentation due to both mismatches between object sizes and size classes (internal fragmentation), and distribution of live objects among different size classes (external fragmentation). Some MS implementations occasionally perform mark-sweep-compaction to limit external fragmentation, but we do not explore that option here. Since MS is a whole heap collector, its maximum pause time is poor and its performance suffers from repeatedly tracing objects that survive many collections.

4.6 FG-MS and VG-MS: Generational Copying/Mark-Sweep

These hybrid generational collectors use a fixed-size (FG-MS) or a variable-size (VG-MS) nursery and a mark-sweep policy for the older generation. Both allocate using a bump pointer and when the nursery fills up, trigger a nursery collection. FG-MS and VG-MS are use the same code as FG and VG, except that semi-space mature space is replaced with a MS mature space. Thus the write barrier, nursery collection, and nursery allocation policies and mechanisms are identical to those for FG and VG. In both cases, the test for the heap being full must accommodate space for copying a entire nursery full of survivors.

If after a nursery collection the heap is full, these collectors perform a full heap MS collection over the old space. By exploiting the generational hypothesis, they mitigate the drawbacks of MS for throughput and average pause times, but full heap collections drive up maximum pause times.

4.7 VC-MS: Copying/Mark-Sweep

This whole heap collector works similarly to VG-MS, except that it has no write barrier, and collects the entire heap every time. It triggers a collection when the heap is full, copies nursery survivors to the old mark-sweep space, and reduces the nursery size by the size of the old space in addition to holding a matching copy reserve. It is an interesting hybrid because by comparing it with MS and VG-MS we can pull apart the benefit of copying young objects as opposed to delaying the collection of the old space.

4.8 RC: Reference Counting

The pure deferred reference-counting collector [25] organizes the heap using the same free-list allocator as MS. It defers counting references from the registers, stacks, and class variables. The write barrier remembers other new pointers to objects in an increment buffer, and over-written pointers to objects in a decrement buffer. Our implementation collects after a fixed amount of allocation. This collector uses Bacon et al.'s trial deletion algorithm to find dead cycles [12]. Collection time is proportional to the number of dead objects, but the mutator load is significantly higher than in the generational collectors or other full heap collectors since it records one or two entries for every heap pointer store.

4.9 Implementation Details

This section adds a few more implementation details about the write barriers, allocation paths, size classes, reference counting header, the large object space, and the boot image.

For all of the generational collectors, we inline the write-barrier fast path which filters out stores to nursery objects and thus does not record most pointer updates, i.e., ignores between 93.7% to 99.9% of pointer stores. The slow path makes the appropriate entries in the remembered set. Since the write barrier for RC is unconditional, it is fully inlined but the increment and decrement remembering mechanisms are forced out of line to minimize code bloat and compiler overhead [17]. The full heap collectors, SS, MS, and VC-MS, have no write barrier. We also inline the fast path for the allocation sequence.

For the copying collectors, the inlined sequence consists of incrementing a bump pointer and testing it against a limit pointer. If the test fails (failure rate is typically 0.1%), the allocation sequence calls an out-of-line routine to acquire another block of memory.

For the free-list collectors (MS and RC), the inline allocation sequence consists of establishing the size class for the allocation (which for scalars is statically evaluated using the optimizing compiler), and removing a free cell from the appropriate free list, if such a cell is available. If there is no available free cell, the allocation path calls the slow path to move to another block, or if there are no more blocks of that size class, acquire a new block.

We have a two word (8 byte) header for each object which contains a pointer to the TIB (type information block located in the immortal space), hash bits, lock bits, and GC bits. We can correctly implement a one word header for MS collectors, but have not yet done so. Bacon et al. found that a one word header yields an average of 2-3% improvement in overall execution. [9] The RC

collector has an additional word (4 bytes) in the object headers to accomodate the reference count.

For our free-list allocators, we use a range of size classes similar to, though less extensive than, the Lea allocator [33]. We select 51 size classes that attain a worst case internal fragmentation of 1/8. The size classes are 4 bytes apart from 8 to 63, 8 bytes apart from 64 to 127, 16 bytes apart from 128 to 255, 32 bytes apart from 256 to 511, 256 bytes apart from 512 to 2047, and 1024 bytes apart from 2048 to 8192. Small, word-aligned objects get an exact fit—in practice, these are the vast majority of all objects. All objects 8KB or larger get their own block.

JMTk implements the large object space (LOS) as follows. For pure free-list allocators (MS and RC), it just allocates large objects directly. For hybrid collectors with a free-list in the older generation, we allocate (*pretenure*) objects that are 64K or larger directly onto the free-list in the old space. For SS, FG, and VG, we add a MS space only for these objects. During full heap collections, we scan and collect the large objects. We do not apriori reserve space for the LOS, but allocate it on demand.

The boot image contains various objects and precompiled classes necessary for booting the Jikes RVM, including the compiler, classloader, the garbage collector, and other essential elements of the virtual machine. None of the Jikes RVM collectors *collect* the boot image objects. Jikes RVM's tracing collectors (including SS, VG, FG, MS, VG-MS, and FG-MS) trace through the boot image objects whenever they perform a full heap collection.

5. Methodology

This section briefly describes the Jikes RVM, our experimental platform, and some key characteristics of our benchmarks.

5.1 The Jikes RVM

We use the Jikes RVM (formerly known as Jalapeño) for our experiments with JMTk. The Jikes RVM is a high-performance VM written in Java with an aggressive optimizing compiler [1, 2]. The Jikes RVM offers three compiler choices: *baseline*, a quick non-optimizing compiler for all methods; *optimizing*, an aggressive optimizing compiler for all methods; and *adaptive*, which initially uses baseline and adaptively recompiles hot methods with the optimizing compiler [4]. The adaptive compiler uses sampling to select optimization candidates, and thus tends to make slightly different choices for each execution which are influenced by changes in the collector and write barrier. This non-determinism can make the adaptive compiler a difficult platform for any detailed study, but we use it for this study because it places the most realistic load on the system. This introduces variations in the load on the garbage collector because the write barrier for each collector is part of the runtime system as well as the program and induces both different mutator behavior and collector load [17]. The Jikes RVM can be configured with various levels of ahead-of-time compilation. A minimal configuration only precompiles those classes essential to bootstrapping the VM (which does not include the optimizing compiler). We turn off assertion checking for our experiments. We use the configuration which precompiles as much as possible, including key libraries and the optimizing compiler (the *Fast* build-time configuration).

5.2 Experimental Platform

We perform all of our experiments on 2 GHz Pentium 4, with 8KB L1 data cache, a 12K L1 instruction cache, a 512KB unified L2 on-chip cache, and 1GB of memory running Linux 2.4.18. We run each benchmark at a particular parameter setting five times and use the fastest of these. The variation between runs is low, and we believe this number is the least disturbed by other system factors and the natural variability of the adaptive compiler.

		JMTk			Watson			
benchmark	collector	min heap *MB*	alloc *MB*	alloc/ min	immortal *MB*	large *MB*	SS *MB*	MS *MB*
compress	VG-MS	13	158	12	1	13	4	3
jess	VC-MS	6	287	48	1	6	8	7
raytrace	VC-MS	11	154	14	1	6	29	9
db	VC-MS	17	93	5	1	7	20	13
javac	VC-MS	18	253	14	1	6	24	23
mpegaudio	VC-MS	7	42	6	1	6	6	5
mtrt	VG-MS	19	164	9	1	6	38	12
jack	VG-MS	12	257	21	1	6	7	9
pseudojbb	VG-MS	7	323	46	1	7		49

Table 1: Benchmark Characteristics Using the FastAdaptive Compiler

5.3 Benchmarks

Table 1 shows key characteristics of each of our benchmarks. We use seven taken from the SPEC JVM benchmarks, and pseudojbb, a slightly modified variant of SPEC JBB2000 [36, 37]. Rather than running for a fixed time and measuring transaction throughput, pseudojbb executes a fixed number of transactions. This modification makes it possible to compare running times under a fixed garbage collection load. The collector column indicates which collector works in the smallest heap, and that heap size is listed in the *min heap* column. The minimum heap size is inclusive of the memory requirements of the adaptive system compiling the benchmark. Notice VC-MS generally executes in the tightest heap, however, it never performs particularly well. The *alloc* column in Table 1 indicates the total number of bytes allocated for each benchmark. The fourth column indicates the ratio between total allocation and MS minimum heap size, giving an indication of the garbage collection load for each benchmark. This ratio shows these are reasonable, but not great benchmarks for garbage collection experiments. Real Java benchmarks with ratios greater than 100 would be welcome. Section 7 describes remaining columns in the table.

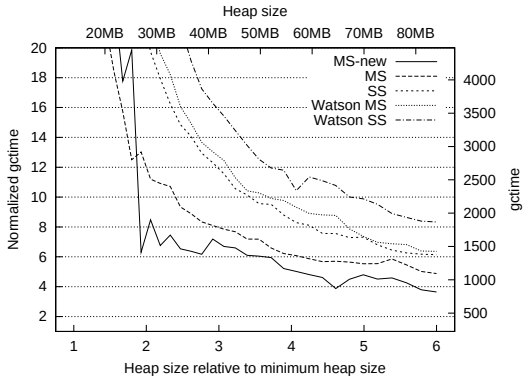
6. Results

In the following sections, we first compare our implementations of SS, and MS with the original Jikes RVM collectors called the *Watson collectors* to illustrate that our collector design has not come at a performance penalty. Our measurements include garbage collection, mutator, and total time. The results show that the flexibility of dynamically partitioning the heap and reuse of highly optimized components can result in performance improvements for JMTk as compared to the highly tuned Watson collectors.

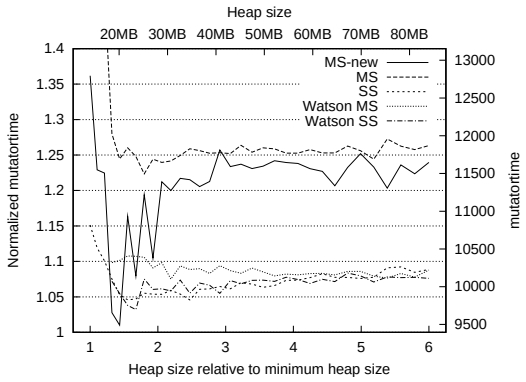
We then systematically compare JMTk collectors, beginning with the full heap collectors: SS, MS, VC-MS, and RC. We find, for example, that SS performs worse than MS in a tight heap because of its need for a copy reserve that causes more frequent collections. In addition, the mutator cost of RC causes it to perform much more poorly than the other benchmarks. For these benchmarks, the generational collectors significantly outperform the full heap collectors on all metrics. Section 9 presents the performance results for the generational collectors: FG, VG, FG-MS, and VG-MS. We find that the flexibly-sized nursery attains better performance than a fixed-size nursery for both a copying and MS older generation, and that for our benchmarks, VG and VG-MS perform very similarly.

7. JMTk versus Watson Collectors

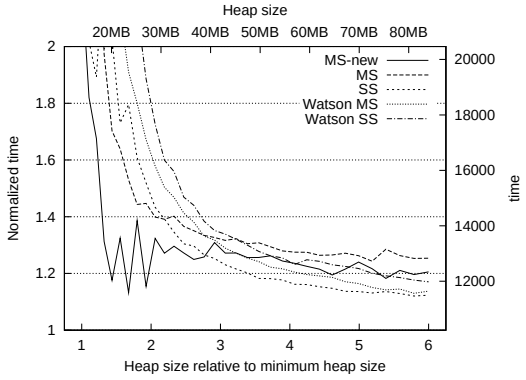
This section compares the SS, MS, and new-MS (an improved, but in-progress MS collector described below) JMTk collectors to the



(a) GC time



(b) Mutator time



(c) Total time

Figure 1: JMTk vs. the Watson Collectors

equivalent well-tuned Watson collectors, Watson-SS and Watson-MS. Although the collectors are equivalent in spirit, a few differences in the implementations account for the variations that appear in Figure 1. These results are collected as the heap size varies from the minimum shown in Table 1 to 6 times the minimum at 32 different points. We use the geometric mean over our benchmarks and measure garbage collection, mutator (which includes allocation and adaptive compilation time, as well as the application), and total time. We normalize these graphs to the best performing collector in our study which is almost always one of the generational collectors. We also measure the number of collections for all our experiments, but do not present the graphs since the collection time is essentially perfectly predicted by the number of collections.

A few differences in the monolithic implementations of the Watson collectors are as follows. Both JMTk and Watson directly manage objects larger than 8 KB with a large object space (LOS). These collectors trace the LOS on every collection. Watson's LOS is a first-fit algorithm with page alignment. It does not maintain a free list. On a request, it scans through the LOS memory until it finds sufficient contiguous free pages. We use a free list (see Section 4.9).

The Watson collectors statically divide the heap into small, large, and immortal objects based on command line parameters. We experimentally determined the smallest possible parameter for the large and immortal space to maximize the amount available for other (most) objects for these experiments. These settings enable us to make fair comparisons by giving the Watson collectors the best possible command-line parameters, and are presented in Table 1. Comparing their total to the JMTk minimum, shows that this feature alone is costly in space. A command line parameter sets the total heap size in JMTk, and then JMTk dynamically determines the divisions into the three spaces (and others depending on the collector) based on program allocation. JMTk thus enjoys a space advantage during the periods that the program is not using the maximum in the large and immortal space, since the Watson partitions are statically sized.

Algorithmically, the SS and Watson-SS are essentially the same. Their implementations of the inlined allocation sequence are also very similar and they thus perform very similarly with respect to mutator performance, as shown in Figure 1(b). Figure 1(a) and (c) show that the collection phase and consequently the total time for SS is actually slightly faster than the Watson-SS. Currently, the scan and copy implementation in JMTk is slower than in the Watson collectors as measured by the number of bytes scanned or copied per second by about 5%. However, because JMTk only allocates space to large objects when they are in use, it can and does use this space when it is available for other heap objects, and this feature results in fewer collections and the performance advantage we see for SS. We expect that most of this time can be recovered given sufficient tuning so that JMTk's overall advantage will be even greater.

Algorithmically, the Watson MS collector is similar to MS and new-MS, but it uses different size classes. It uses power of two, except below 32 bytes it adds a few: 8, 12, 16, 20, 32, 64, 128, ..., 8192, with worst case internal fragmentation of $1/2$ of the heap. Since most objects are less than 64, this worst case is unlikely. However our size classes get a perfect fit on all objects less than 64 bytes and have a worst case fragmentation of $1/8$ (*cf.* Section 4.9). Because the Watson-MS collector has a one word header, it enjoys a runtime advantage of on average 2% for our benchmarks over the two word header in JMTk. (We have not yet implemented this optimization in JMTk.)

Another optimization Watson-MS implements is lazy sweeping, which sets the unused bits, but it does not actually weave through free memory constructing the free list for a block until an allocation fits in the block. This optimization eliminates the reclamation of free objects at collection time which reduces collector processing time to a function of the live data, and eliminates accesses that are likely to be costly due to poor locality until the mutator would have used them anyway. This feature and a more streamlined allocation path are the major differences between MS and new-MS. This difference accounts for the collection time advantage new-MS

enjoys over MS, since they perform exactly the same number of collections.

We very recently got new-MS working which is why all of the other experiments use MS as the base implementation for the hybrids. For the final paper, we plan to use new-MS in all our results.

Even with these drawbacks, Figure 1(a) and (c) illustrates that both MS and new-MS outperform Watson-MS on collector and total time. We believe that the dynamic specification of the large object space and to a lesser extent the space efficiency of our size classes are the reasons. We do have two space size disadvantages: (1) JMTk headers are twice as big, and (2) all the meta-data for JMTk lives in the heap, whereas Watson-MS puts the meta-data in the boot image. Our measurements of the number of collections confirms the heap occupancy explanation: MS and new-MS generally perform fewer collections than Watson-MS, except in large heaps where all of the collectors perform very few collections. Fewer full heap collections imply that MS is more effectively using its space and the dominate effect in a small heap is the dynamic partitioning of the heap by JMTk.

One interesting point is that MS and new-MS achieve this result even though their allocation paths are not as highly optimized as Watson-MS as can be seen by the significantly worse mutator performance in Figure 1(b). We also confirmed this result in micro-measurements with no compiler activity. When the number of collections is very small in larger heaps, then this effect dominates and Watson-MS slightly outperforms MS on total time. In a moderate to small heap, MS and new-MS outperform Watson-MS as shown in Figure 1(c).

These results show that our basic mechanisms can probably be further optimized, but that our dynamic algorithms for heap partitioning results in well performing collectors. Now we further explore the tradeoffs that different memory management policies make and how they exploit object demographics.

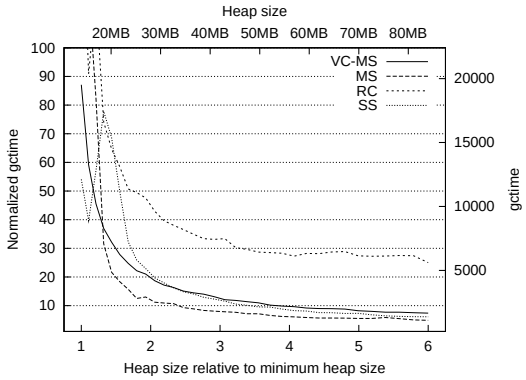
8. Full Heap Collector Bakeoff

This section compares the JMTk full-heap (i.e., non-generational) collectors: SS, MS, VC-MS, and RC. As before, we compare the collection, mutator, and total time across a range of heap sizes for the collectors, and normalize to the best performing collector for the benchmark which is usually a generational collector. Figure 2 gives the geometric mean of the normalized execution times for all benchmarks, and Figures 5, 6, and 7 show garbage collection, mutator, and total time for the individual programs.

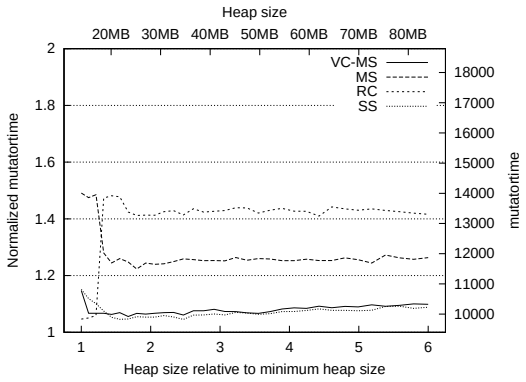
The worst performing collector on all metrics is RC in which we collect every 4MB of allocation. Mutator overhead is much worse because every pointer store results in a buffered increment and decrement. The collector time is not as low as one might expect, but is consistent with Bacon et al. [10] in which they need a separate processor to tolerate reference counting and the latency of the trial deletion phase to detect dead cycles. Trial deletion traces objects for which a decrement does not result in a zero reference count, and is expensive.

Figure 2(a) shows that MS has the best collection times. It achieves this result because it uses space more efficiently than SS and VC-MS. Both SS and VC-MS require copy reserve space. The copy reserve for SS is half the heap, and for VC-MS, it is equivalent to the dynamically-sized nursery. In the whole heap context, this extra space overhead causes more frequent collections and is the reason MS is better since it utilizes more of the heap and collects less frequently in our measurements. SS and VC-MS have similar collection times in general with VC-MS slightly outperforming SS when the heap size is less than 2.5 since it does not need a copy reserve for long-lived objects and at larger heaps this advantage is less important. Asymptotically at large heap sizes, the MS collector outperforms SS by 40% and SS outperforms VC-MS by about 15%. At the other extreme with heap sizes around 2, the MS collector is twice as fast as either SS and VC-MS due to lower collection frequency.

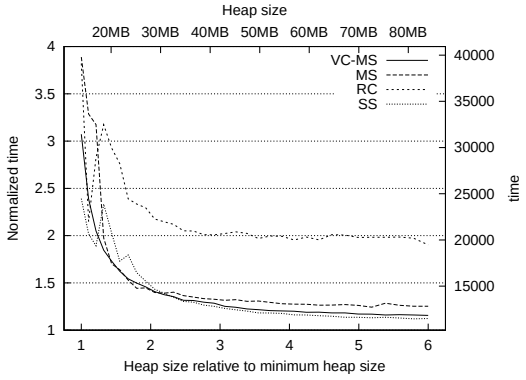
On the other hand, Figure 2(b) shows that both SS and VC-MS have faster mutation time than MS by about 20-30%. The advantage is consistent across all heap sizes. Bump-pointer allocation



(a) GC time



(b) Mutator time



(c) Total time

Figure 2: Full Heap Collectors

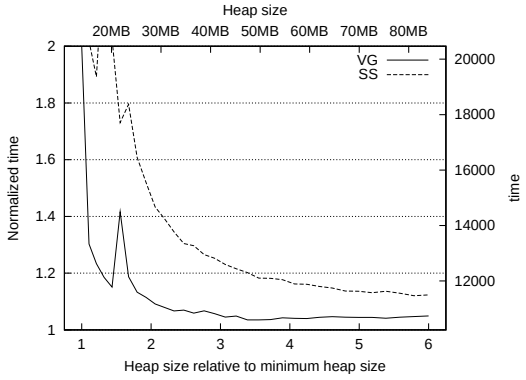


Figure 3: Generational versus Full Heap Collectors

is simply the fastest possible at execution time. Finding the right size block, and updating the inuse and free list clearly has a non-trivial cost. There is a modest degradation in mutator performance of about 3% for SS and VC-MS as we move from a heap size of 2 to 6 for _227_mtrt, _202_jess, and _222_mpegaudio (see Figure 6). The large heap size is degrading locality.

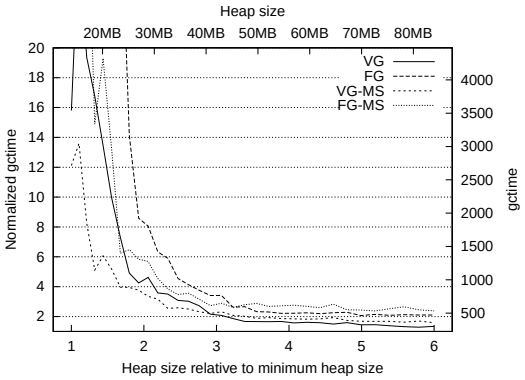
Finally, we consider overall performance time in Figure 2(c). On the whole SS and VC-MS are similar with the latter having an advantage at the smallest heap size and the former an advantage that grows to 5% at large heap sizes. For instance, VC-MS performs much better in the smallest heap on _209_db, _213_javac, and _205_raytrace as shown in Figure 7. The difference between SS and MS is more significant. Below a heap size of about 2.75, the time spent in GC dominates and MS is arbitrarily faster since the survival rate in a full heap collection is low. As the heap size varies from 3 to 6 though, SS's advantage grows from 3% to 9% over MS because its faster allocation time starts to dominate total performance.

The tradeoff between SS and MS is between better locality and less frequent garbage collections. Since the effect of garbage collection time depends on heap sizes, the overall advantage goes to MS at small heaps and to SS at larger heaps. Overall, VC-MS does not outperform either SS nor MS.

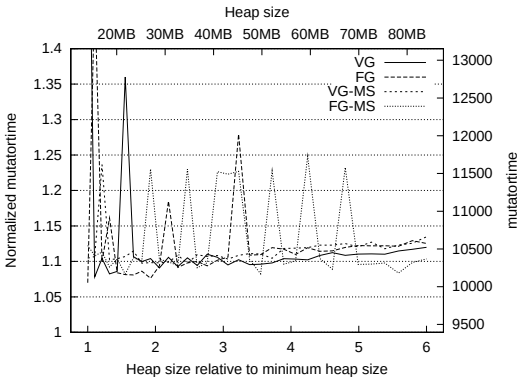
When we look at the collection times of the individual benchmarks, the geometric mean predicts individual program performance. The lone exception is pseudojbb where SS is occasionally faster than VC-MS. Unlike collection time, the effect of collector policy on mutator times is dramatic on all but _201_compress and _222_mpegaudio which mainly allocate large objects. Of the remaining benchmarks, MS is worse than SS and VC-MS by about 15% to 35%. Finally, when we look at total time, the collector policy has little effect on _201_compress, _222_mpegaudio, and _213_javac. Unusually, for the _209_db benchmark, VC-MS, which is usually like SS, behaves like MS. The advantage of SS and VC-MS over MS is most significant in _202_jess and _227_mtrt at 20%.

9. Generational Collector Bakeoff

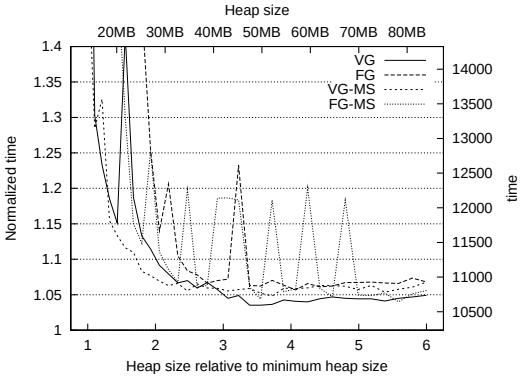
Before comparing the generational collectors in detail, we confirm that generational collectors, in general, outperform the full-heap collectors. For this comparison consider Figure 3 which compares SS and VG-SS. Across the whole range of heap sizes, VG-SS outperform SS (the overall non-generational winner) by 20% In particular, VG-SS wins in every benchmark except for _209_db where it is worse by up to 5% when the heap size is over 4 times the minimum and both collectors perform very few collections.



(a) GC time



(b) Mutator time



(c) Total time

Figure 4: Generational Heap Collectors

This section compares four generational collectors: VG-SS, FG-SS, VG-MS, and FG-MS. All collectors use a copying nursery but the mature spaces are either copying (VG-SS and FG-SS) or mark-sweep (VG-MS and FG-MS). Within each pair, the first one uses a variable-size nursery while the second uses a fixed-size nursery.

Figure 4(a) compares collection time for the four generational collectors. The collection times are stable with the VG-SS and VG-MS outperforming their fixed-size counterparts. Other research has

established that for these benchmarks, neither a third generation nor another nursery size changes the relative performance of VG and FG [16, 38], and we believe the same trend will hold for VG-MS and FG-MS on different nursery sizes.

On collection time, VG-SS does better than VG-MS when the heap size is over 3. As usual, the purely copying collectors suffer from more frequent collections at tight heap sizes due to the space overhead of a copy reserve (see Section 8). When we examine the collection times for the individual benchmarks in Figure 8, they are in agreement with the geometric mean. In some cases, such as _201_compress, there is a distinct staircase effect stemming from the discrete nature of garbage collection. A few unusual cases are worth mentioning. In _209_db, the fixed-size nursery spends less time in collection and the variable-size collectors actually spend more overall time in collection as the heap size increases. In pseudojbb, the advantage of VG over FG is unusually large at a factor of four.

Figure 4(b) shows the geometric mean of the mutator time for the various collectors, and Figure 4(c) shows the total time. These graphs have spikes that do not really exist for FG-SS and FG-MS, as shown by the the individual program data in Figures 9 and 10 but result from missing data points that defeat our geometric mean calculation. We will fix this problem in the final paper.

We see greater variations for mutator time from the different generational collectors, than for the collection time. No collector always dominates mutator time as shown in Figure 9. Since all the collectors use bump-pointer allocation, the variations in mutator time are usually small, as expected. For _209_db, the copying variants have a 15% advantage over the MS variants which we believe is due to improved locality with copying. _209_db is very sensitive to changes in its data layout. This effect is clear but less pronounced in _227_mtrt at 5%. In _222_mpegaudio, the FG variants win by 5%. Curiously, in _213_javac, FG-SS is the worst but FG-MS is the best. Again we see degradations in mutator time as the heap grows for _213_javac, _222_mpegaudio, and _227_mtrt when there are not enough collections to pack the live data in the old space well.

The generational collectors' overall performance as a function of heap size (Figure 10) for each benchmark is mainly dictated by collector time. VG-SS is often the best in large heaps, but VG-MS does better in tight heaps. The overall results are not encouraging for constrained memory. Even with generational collectors, memory management costs are prohibitive. Garbage collection algorithms still trade space for time, and that tradeoff needs to be better balanced.

10. Future Work

While JMTk has met its initial design goals, there are several areas where further work is necessary. First, the portability of JMTk has yet to be tested and the work to port it to another system is ongoing. Second, the parallelism of JMTk is not tested and preliminary evidence shows that, at least in some cases, the scalability is noticeably worse than for the Watson collectors. Third, while the design of JMTk is general, the code is somewhat immature and further tuning, including the incorporation of the one-word object model, is necessary.

11. Conclusion

JMTk is a portable, extensible, composable memory management toolkit for Java in which similar collector algorithms have performance comparable to that of monolithic, hand-tuned collectors. The flexible design of JMTk makes it simple to perform fair comparisons of competing algorithms because they share implementations of all common components. Once the basic mechanisms are in place, a collector is easy to implement. We confirm exhaustively, across a range of heap sizes and benchmarks, several empirical facts that are mostly but not universally held by the community. First, a copying collector has better locality than a mark-sweep collector so that when the mutator time is dominant (when the heap size is 2.5 times the minimum heap size), the copying collector provides better overall performance. Second, generational collectors outperform full-heap collectors by 20% on average and, in the rare cases (one out of nine benchmark) loses by 2% when the heap size is larger than 4 times the minimum heap size. Third, among the generational collectors, the variable-size nursery collectors outperforms the fixed-size nursery variants by 2%.

12. REFERENCES

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA '99, Denver, Colorado, November 1-5, 1999*, volume 34(10) of *ACM SIGPLAN Notices*, pages 314–324, Oct. 1999.

[2] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1):211–238, February 2000.

[3] A. W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[4] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA'00 ACM Conference on Object-Oriented Systems, Languages and Applications, Minneapolis, MN, USA, October 15-19, 2000*, volume 35(10) of *ACM SIGPLAN Notices*, pages 47–65, October 2000.

[5] C. R. Attanasio, D. F. Bacon, A. Cocchi, and S. Smith. A comparative evaluation of parallel garbage collectors. In *Languages and Compilers for Parallel Computing, 14th International Workshop, LCPC 2001, Cumberland Falls, KY, USA, August 1-3, 2001*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

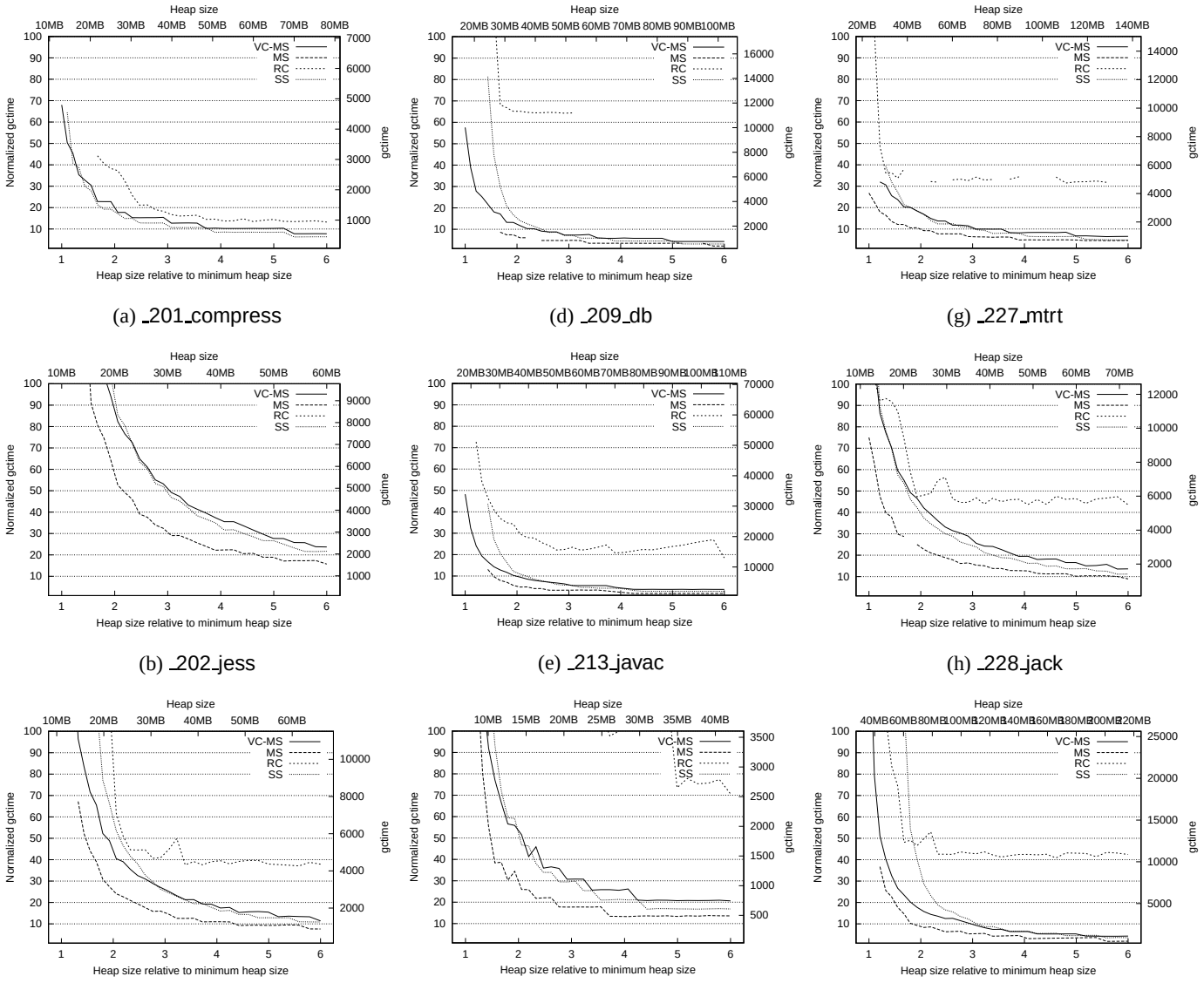
[6] G. Attardi and T. Flagella. A customizable memory management framework. In *Proceedings of the USENIX C++ Conference*, Cambridge, Massachussetts, 1994.

[7] G. Attardi, T. Flagella, and P. Iglio. A customizable memory management framework for C++. *Software Practice & Experience*, 28(11):1143–1183, 1998.

[8] H. Azatchi and E. Petrank. Integrating generations with advanced reference counting garbage collectors. In *International Conference on Compiler Construction*, Warsaw, Poland, Apr. 2003. To Appear.

[9] D. Bacon, S. Fink, and D. Grove. Space- and time-efficient implementations of the java object model. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 111–132. ACM Press, June 2002.

[10] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Languages*



(a) _201_compress (d) _209_db (g) _227_mtrt

(b) _202_jess (e) _213_javac (h) _228_jack

(c) _205_raytrace (f) _222_mpegaudio (i) pseudojbb

Figure 5: GC Time for Full Heap Collectors

Design and Implementation (PLDI), Snowbird, Utah, May, 2001, volume 36(5), June 2001.

[11] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the Thirtith Annual ACM Symposium on the Principles of Programming Languages*, pages 285–294, New Orleans, LA, Jan. 2003.

[12] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In J. L. Knudsen, editor, *Proceedings of 15th European Conference on Object-Oriented Programming, ECOOP 2001, Budapest, Hungary, June 18-22*, volume 2072 of *Lecture Notes in Computer Science*, pages 207–235. Springer-Verlag, 2001.

[13] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Nov. 2000.

[14] E. D. Berger, B. G. Zorn, and K. S. McKinley. Building high-performance custom and general-purpose memory allocators. In *Proceedings of the SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 114–124, Salt Lake City, UT, June 2001.

[15] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *ACM Conference Proceedings on Object–Oriented Programming Systems, Languages, and Applications*, pages 1–12, Seattle, WA, Nov. 2002.

[16] S. M. Blackburn, R. E. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*,

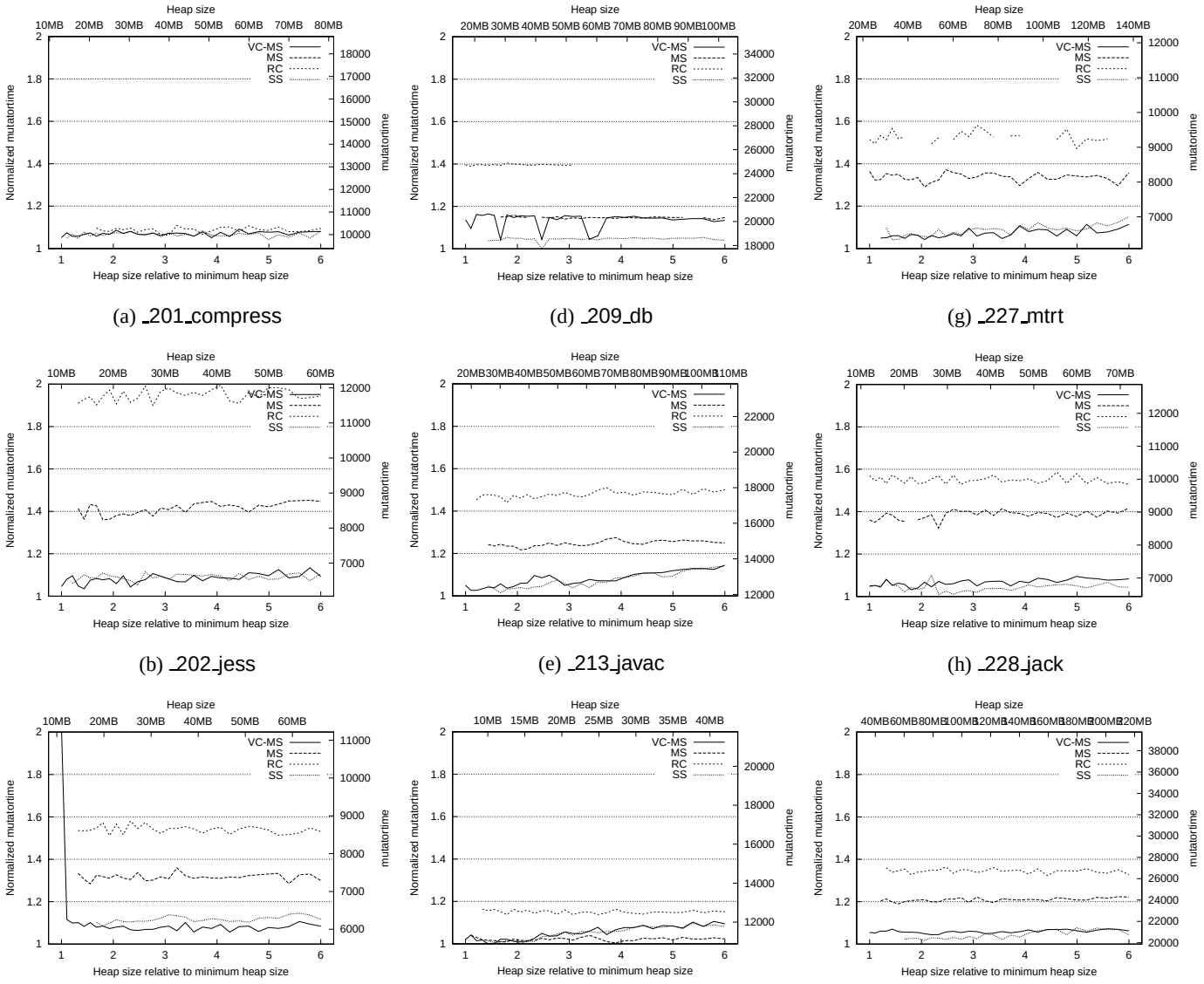


Figure 6: Mutator Time for Full Heap Collectors

PLDI'02, Berlin, June, 2002, volume 37(5) of *ACM SIGPLAN Notices*, Berlin, Germany, June 2002.

[17] S. M. Blackburn and K. S. McKinley. In or out? Putting write barriers in their place. In *Proceedings of the Third International Symposium on Memory Management, ISMM '02, Berlin, Germany*, volume 37 of *ACM SIGPLAN Notices*. ACM Press, June 2002.

[18] S. M. Blackburn, S. Singhai, M. Hertz, , K. S. McKinley, and J. E. B. Moss. Pretenuring for java. In *ACM Conference Proceedings on Object–Oriented Programming Systems, Languages, and Applications*, pages 342–352, Tampa, FL, Oct. 2001. ACM.

[19] H.-J. Boehm. Space efficient conservative garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–206, 1993.

[20] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) / Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[21] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001*, volume 36(11) of *ACM SIGPLAN Notices*, Tampa, Florida, USA, Nov. 2001.

[22] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, Nov. 1970.

[23] J. Dean, G. DeFouw, D. Grove, V. Litinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented

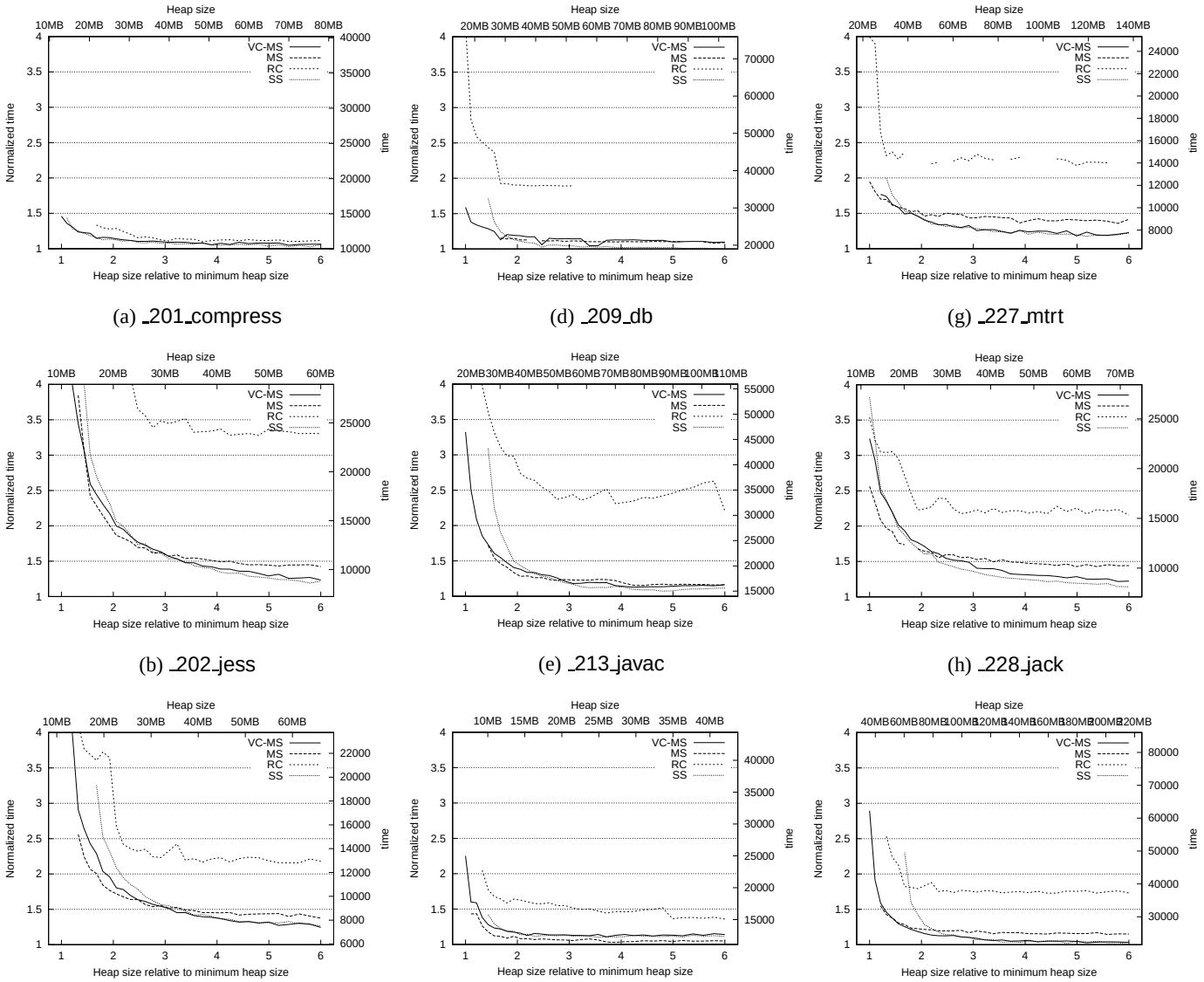


Figure 7: Total Time for Full Heap Collectors

languages. In *ACM Conference Proceedings on Object–Oriented Programming Systems, Languages, and Applications*, pages 83–100, San Jose, CA, Oct. 1996.

[24] D. L. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. *Software Practice & Experience*, 24(6):527–542, June 1994.

[25] L. P. Deutsch and D. G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[26] E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens. On-thefly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, September 1976.

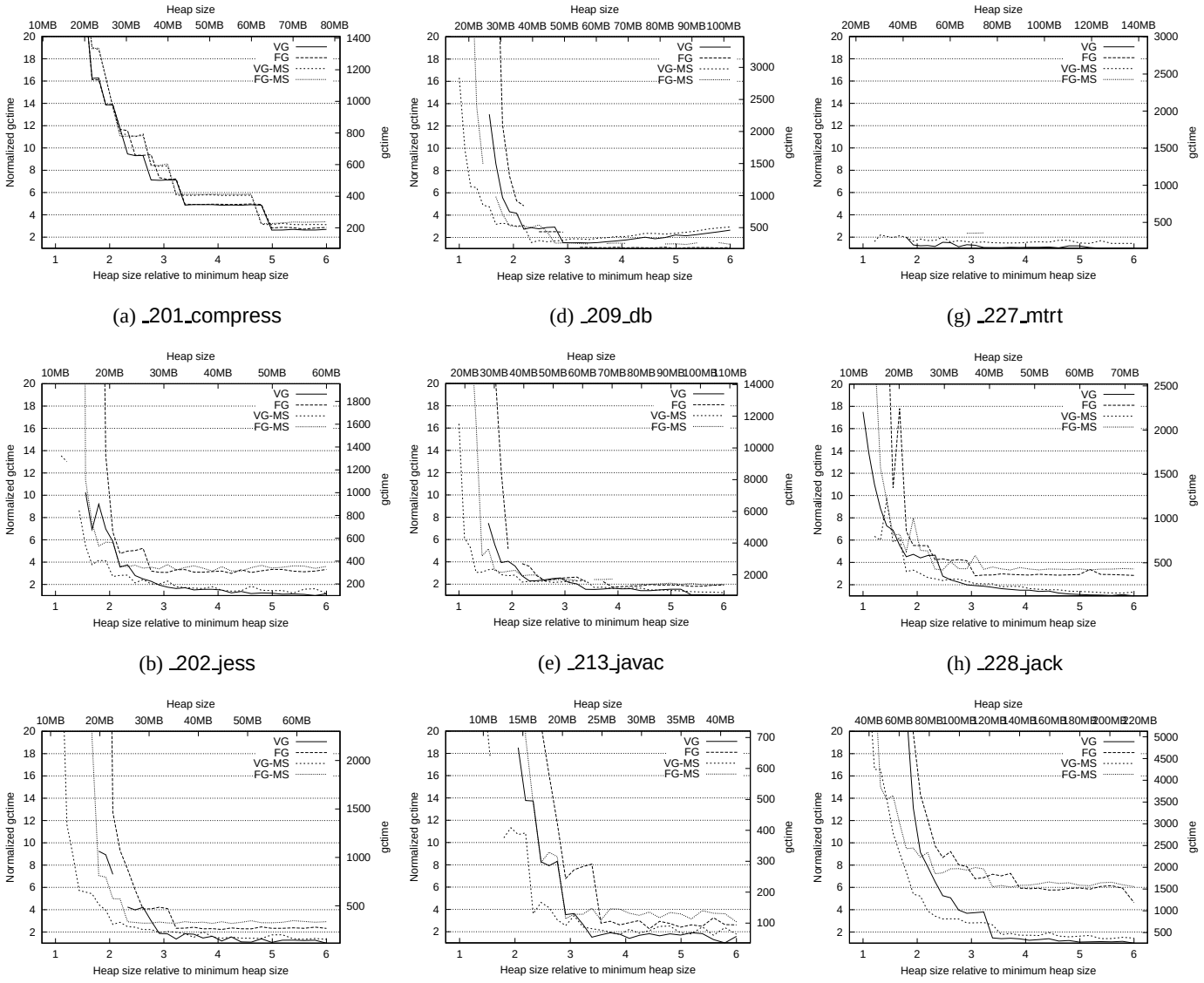
[27] A. Diwan, J. E. B. Moss, and R. L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 273–282, San Francisco, California, June 1992.

[28] R. Fitzgerald and D. Tarditi. The case for profile-directed selection of garbage collectors. In T. Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, pages 111–120, Minneapolis, MN, Oct. 2000.

[29] M. W. Hicks, J. T. Moore, and S. Nettles. The measured cost of copying garbage collection mechanisms. In *International Conference on Functional Programming*, pages 292–305, 1997.

[30] A. L. Hosking and R. L. Hudson. Remembered sets can also play cards, Oct. 1993. Position paper for OOPSLA '93 Workshop on Memory Management and Garbage Collection.

[31] R. E. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July



(a) _201_compress (d) _209_db (g) _227_mtrt

(b) _202_jess (e) _213_javac (h) _228_jack

(c) _205_raytrace (f) _222_mpegaudio (i) pseudojbb

Figure 8: GC Time for Generational Collectors

1996.

[32] J. Kim and Y. Hsu. Memory system behavior of Java programs: methodology and analysis. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, Santa Clara, California, June 2000.

[33] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 1997.

[34] Y. Levanoni and E. Petrank. An on-the-fly reference counting garbage collector for Java. In *ACM Conference Proceedings on Object–Oriented Programming Systems, Languages, and Applications*, pages 367–380, Tampa, FL, Oct. 2001.

[35] H. Lieberman and C. E. Hewitt. A real time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[36] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[37] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.

[38] D. Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, 1999.

[39] D. Stefanović, M. Hertz, S. M. Blackburn, K. McKinley, and J. Moss. Older-first garbage collection in practice: Evaluation in a java virtual machine. In *Memory System Performance*, Berlin, Germany, June 2002.

[40] D. Stefanović, K. McKinley, and J. Moss. Age-based garbage collection. In *ACM Conference Proceedings on Object–Oriented Programming Systems, Languages, and Applications*, Denver, CO, Nov. 1999.

[41] D. M. Ungar. Generation scavenging: A non-disruptive high

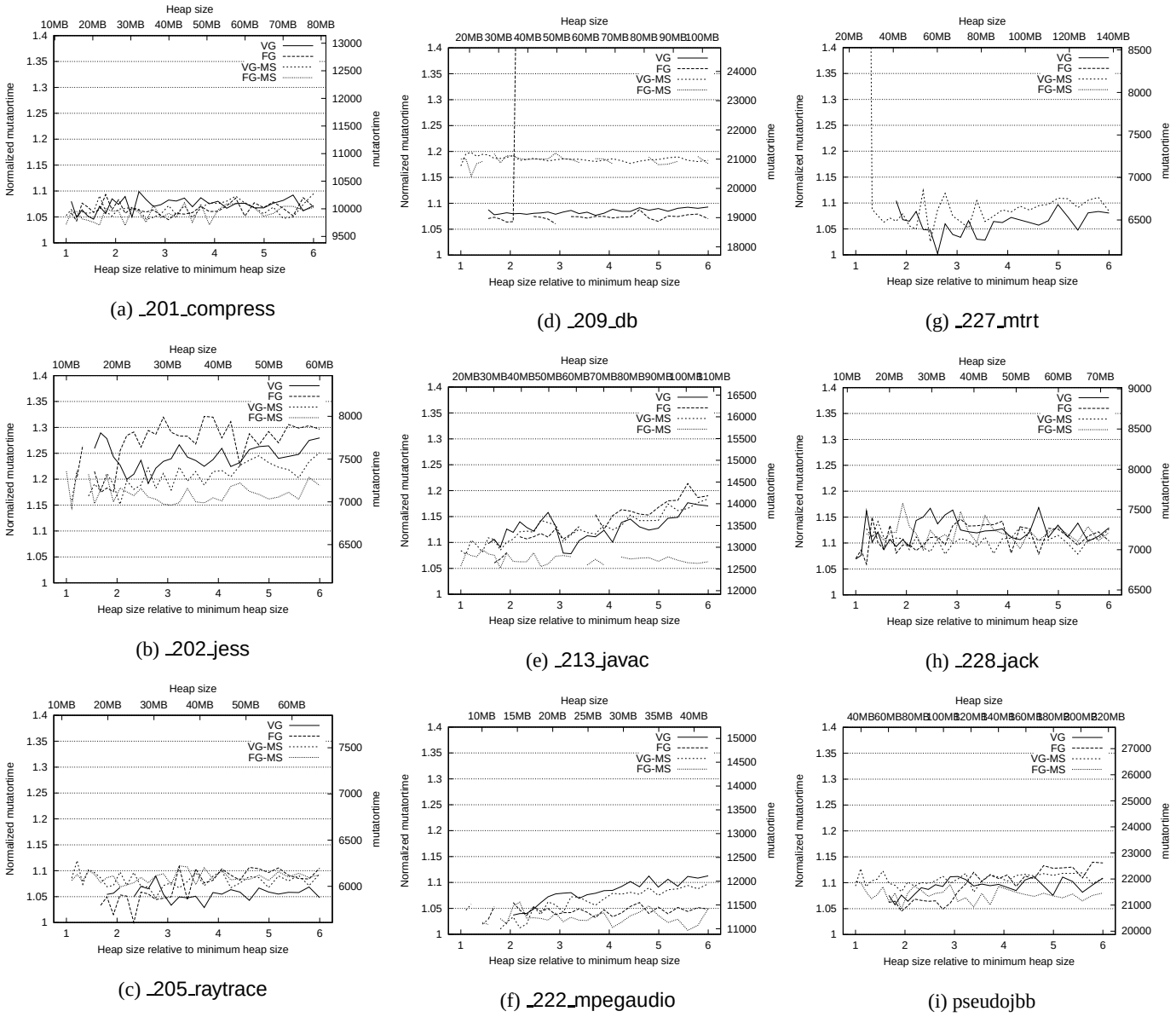


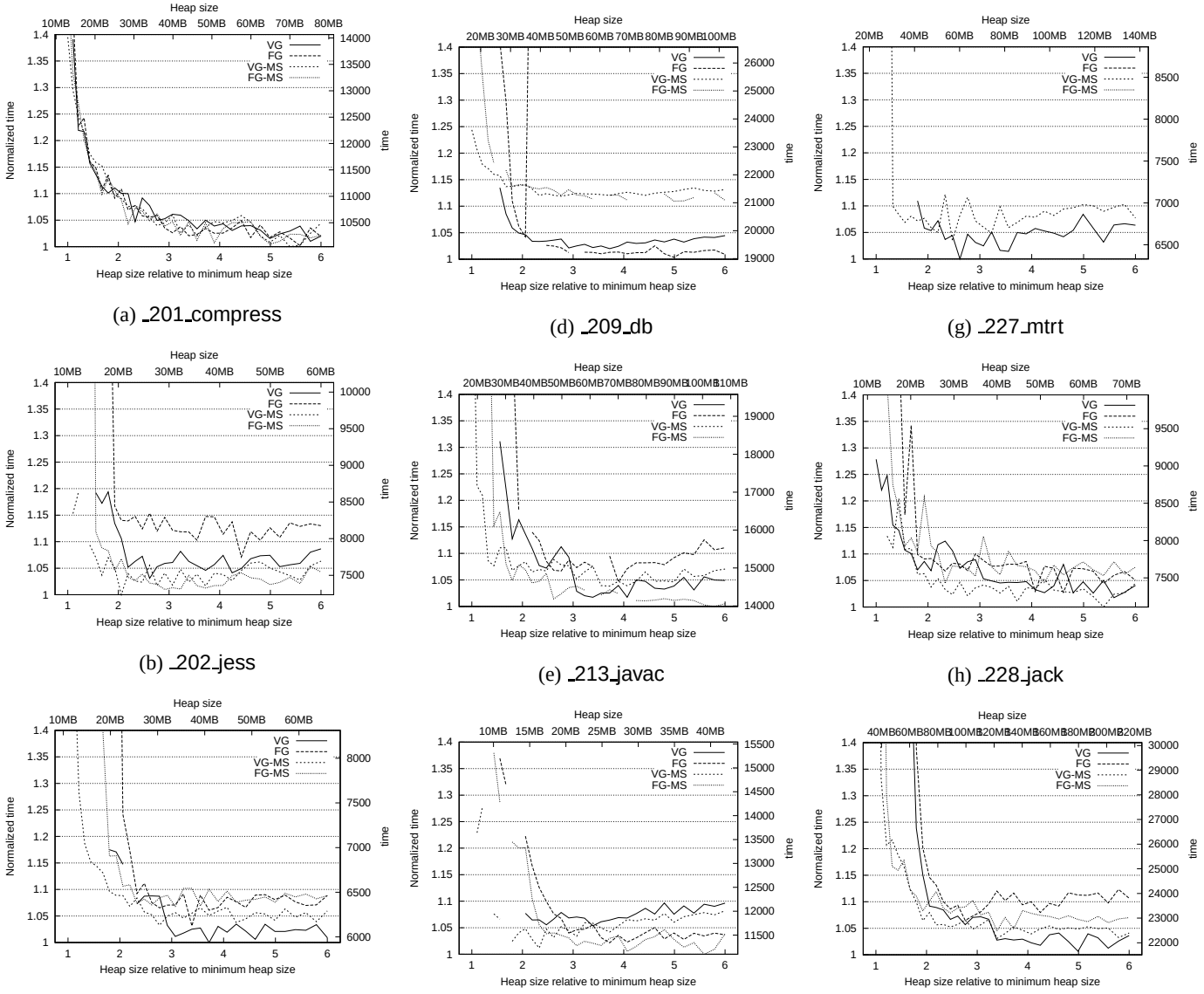
Figure 9: Mutator Time for Generational Collectors

performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.

[42] K.-P. Vo. Vmalloc: A general and efficient memory allocator. *Software Practice & Experience*, 26(3):1–18, 1996.

[43] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In H. Baker, editor, *Proceedings of International Workshop on Memory Management, IWMM'95, Kinross, Scotland*, volume 986 of *Lecture Notes in Computer Science*. Springer-Verlag, Sept. 1995.

[44] B. G. Zorn. The measured cost of conservative garbage collection. *Software Practice & Experience*, 23(7):733–756, 1993.



(a) _201_compress (d) _209_db (g) _227_mtrt

(b) _202_jess (e) _213_javac (h) _228_jack

(c) _205_raytrace (f) _222_mpegaudio (i) pseudojbb

Figure 10: Total Time for Generational Collectors