

# Modern Garbage Collection for Virtual Machines

Sunil Soman

Computer Science Department  
University of California, Santa Barbara  
sunils@cs.ucsb.edu

## 1. INTRODUCTION

Widespread use of Internet computing has made architecture-independent runtime platforms very popular. Examples of such platforms include the Java Virtual Machine [48], and more recently, the Common Language Runtime [20]. Due to the portable nature of applications written for these platforms, explicit memory management is not supported by languages used to implement these applications. In addition, these languages are object-oriented, and as such, applications dynamically allocate heap objects. Consequently, automatic memory management, i.e., garbage collection, is an essential part of virtual machine runtime platforms. However, garbage collection imposes a performance overhead since it must identify and reuse memory that is no longer accessible by the program, *while* the program is executing.

Simplistic garbage collection algorithms cannot be used to provide high performance for virtual machines. Modern garbage collection techniques [53, 34, 16, 38, 15, 11] attempt to achieve high performance by enhancing basic garbage collection algorithms [63]. The goal of most of this prior work has been to provide general-purpose mechanisms that enable high-performance execution across all applications. However, other prior research [5, 32, 66, 55], has shown that the efficacy of a memory management system (the allocator and the garbage collector) is dependent upon application behavior and available resources. That is, no single collection system enables the best performance for all applications and all heap sizes. Our empirical experimentation confirms these findings. Over a wide-range of heap sizes and the 10 benchmarks studied, we found that *every* collector enabled the best performance at least once; *including* a mark-and-sweep and non-generational copying collector, two collectors that are commonly thought of as implementing obsolete technology. As such, we believe that to achieve the best performance, the collection and allocation algorithms used should be specific to both application behavior and heap size.

Existing execution environments enable application- and heap-specific garbage collection, through the use of different configurations (via separate builds or command-line options) of the execution environment. However, such systems do not lend themselves well to next-generation, high-performance server systems in which a single execution environment executes continuously while multiple applications and code components are uploaded by users [40, 33, 52]. For these systems, a single collector and allocator must be used for a wide range of available heap sizes and applications, e.g., e-commerce, agent-based, distributed, collaborative, etc. As such, it may not be possible to achieve high-performance in all cases and selection of the *wrong* GC system may result in significant performance degradation. To address this, we present the design, implementation, and evaluation of

a dynamic GC switching system for JikesRVM, a performance-oriented, server-based, Java virtual machine [2] from the IBM T.J. Watson Research Center. Our switching system facilitates the use of the garbage collector and memory allocator that will enable the best performance for the executing application *and* the underlying resource availability. The system we present is extensible and general; it can switch between many different types of collectors, e.g., semi-space, mark-and-sweep, copying-marksweep, and many variants of generational collection.

To evaluate our system, we implemented two mechanisms: annotation-guided GC selection, and automatic switching. For the former, we identified the best-performing GC for a range of heap sizes for each program, across inputs. We then annotated the programs to identify the collection system to use given different resource constraints. Upon dynamic loading of each application the VM uses the annotation to switch to the appropriate GC given the current maximum available heap size. To implement automatic switching, we employ a simple heuristic that uses maximum heap size and a heap residency threshold to switch during program execution.

Our results show that the cost of switching is equivalent to a garbage collection. In addition, the overhead that our system imposes on application execution performance is 4% on average, for the programs studied. Perhaps more importantly, our system significantly reduces the negative impact of selecting the *wrong* collector (by 19% using annotation-guided selection and over 16% using automatic switching, on average).

In the next section, we will discuss some background for garbage collection including basic garbage collection algorithms. In Section 3, we will discuss modern garbage collection techniques, including the motivation for our current work on application-specific garbage collection, which is the topic of the rest of the paper. In Section 4 we will present an overview of our dynamic GC switching framework. Section 5 describes the different approaches that we employ to dynamically switch between collectors. Section 6 presents and discusses our experimental results. Section 7 discusses other GC switching studies, and Section 8 presents our conclusions.

## 2. BACKGROUND

In this section we will define some garbage collection terms, describe metrics that are used to evaluate garbage collection performance, and examine some basic garbage collection algorithms.

### 2.1 Terminology

We first define a set of garbage collection terms and concepts that we will use in the remainder of this paper.

## Garbage

Garbage is defined as data allocated dynamically by a program that is no longer reachable.

## Mutator

In garbage collection terminology, an application thread is called the mutator, since it mutates or modifies objects.

## Garbage Collector

The term *garbage collector* signifies an automatic memory *reclamation* mechanism, but the data structures for memory management are shared by the reclamation mechanism and the *allocator* [45]. The choice of the allocation algorithm is tied to choice of the reclamation mechanism. The allocator and the collector are therefore implicitly considered to be two components of a garbage collector.

## Garbage Collection Cycle

Any garbage reclamation algorithm can be thought of as consisting of two phases – *garbage detection*, in which live objects are distinguished from garbage, and *garbage reclamation*, in which the space occupied by garbage is freed for use by the application program (the *mutator*). Detection and reclamation constitute a garbage collection cycle.

## Root Set

To detect live objects, a *root set* of references is defined. This consists of static variables, any local (stack-allocated) variables, and general-purpose registers. All objects that are directly or transitively *reachable* from the root set are assumed to be live and cannot be reclaimed. Objects that are unreachable are considered to be garbage, and therefore can be recycled. *Reachability* is a more conservative approach than one used in say, an optimizing compiler, which considers variables to be dead if they are *unused* after a certain point in the program.

## Type Accurate Garbage Collection

Modern programming languages for virtual machine applications are strongly typed, which implies that the compiler fully supports runtime type identification. Consequently, it is possible to accurately identify object references without any need for a conservative approach. A conservative garbage collector is intended for use in languages that have do not provide strong typing, e.g. C and C++. A conservative garbage collector does not know the location of all object references inside an object. As such, such a collector needs to be *conservative* in its identification of object references, i.e., anything that *looks* like a pointer, may be one. On the other hand, garbage collectors designed for virtual machines have full knowledge about an object's type and its internal reference fields, and are therefore, fully *type accurate*.

## Object Header

Modern object oriented languages are dynamically typed, and as such, require a per-object header to hold type information. Virtual Machines typically make use of a single, two-word or three-word object header. The header usually contains the object's identity hash code and GC status information, the format of which is VM dependent. In addition, it also contains a reference to the object's class.

## Uniform representation of Internal Data Structures

The VM typically does not differentiate between internal reflective data structures, like classes and methods and “normal” application objects. This is done to simplify the object model and to enable collection of classes and methods by the same garbage collector that collects application objects.

## Compiler Support for Garbage Collection

Garbage collection needs some cooperation from the compiler to be able to detect live objects. Modern optimizing compilers for strongly-typed languages support garbage collection to a greater extent than those built for languages without a guarantee about types. The former provide precise information about object types to the garbage collector.

Interpreters and compilers differ over when garbage collection can be initiated. GC can be initiated at any point during program execution in case of interpreters, called *GC-anytime*. However, all modern execution environments employ an optimizing compiler, usually with multiple optimization levels. In such a case, garbage collection can only be initiated at certain defined points during execution, called *safe-point* collection. This allows the optimizing compiler to use arbitrarily complex optimizations between safe points, as long as it maintains information that is necessary to locate pointer values at safe points. This information is generated during compilation and is maintained in a per-method data structure called the *garbage collection map* (GC map).

## Finalization

Modern programming languages (Java, C#) allow a *finalizer* to be defined for an object. Finalizers are somewhat similar to destructors in other languages, and allow clean up operations to be performed after the object has been reclaimed, e.g., closing a file. A finalizer is *asynchronously* invoked by the garbage collector after it reclaims the corresponding object. This might lead race conditions, since the order of invoking finalizers is not defined. Also, it might be possible that an object which had been considered to be garbage, is resurrected by a finalizer. Finalization is a major source of automatic memory management problems, but is considered to be a necessary evil.

## 2.2 GC Metrics

We will now describe some metrics that are commonly used to evaluate garbage collection performance.

- **Garbage Collection Time.** The time taken to perform a garbage collection cycle, usually represented as an average or mean for all GC cycles.
- **Pause Time.** The average (called average pause time) or maximum (called maximum pause time) amount of time that an application must wait, while garbage collection is in progress. Even concurrent garbage collectors need application activity to stop (for a short while), while some garbage collection activity is being performed, in order to guarantee correctness.
- **Throughput.** This refers to application performance expressed as the inverse of execution time.
- **Mark/Cons Ratio.** It can be defined as the ratio of the total amount of data copied by the collector to the total amount of data freshly allocated by the program. Mark/cons ratio is considered to be a good indicator of performance for copying collectors. A lower mark/cons ratio is better.

- **Mutation Rate.** The number of objects (or bytes) modified by the mutator in a unit of time (could be seconds, or normalized to the total application execution time).
- **Allocation Rate.** The number of bytes of data allocated by the mutator in a unit of time (could be seconds, or normalized to the total application execution time, or most likely, normalized to the total number of bytes allocated).

## 2.3 Classification

We classify garbage collection techniques into the following broadly defined categories.

- **Stop-the-World.** In stop-the-world collection, the mutator (application program) needs to be paused completely while garbage reclamation is in progress. The garbage collector collects the entire heap in a garbage collection cycle.
- **Incremental.** An incremental collector interleaves reclamation with mutator activity. The garbage collector does not collect the entire heap in one garbage collection cycle, but rather, collects a fixed size area, or *increment* per collection cycle. Incremental garbage collectors are explained in more detail in Section 3.2.
- **Concurrent.** In concurrent collection, the mutator and the collector may modify data simultaneously.
- **Real-Time.** In real-time garbage collection, garbage collection time is bounded by a guaranteed small time constant. A compromise is usually made between throughput and the real-time guarantee. Also, memory overhead is generally higher.
- **Parallel.** Parallel garbage collection implies that multiple collector threads run concurrently to enable reclamation. This is not a separate class of garbage collectors, per se. Even a stop-the-world collector could implement multiple collector threads.

This classification is not very strict. For example, a concurrent collector might require a very short stop-the-world phase (called mostly-concurrent collection). We will now discuss some basic garbage collection algorithms [63]. These are all of the stop-the-world variety, except reference counting, which is inherently incremental.

### 2.3.1 Reference Counting

In reference counting, an integer count is maintained per live heap object, usually in the object header word. The count is incremented whenever a reference to the object is created, and decremented whenever a reference to the object is deleted. A count of zero indicates that the object is garbage and can be reclaimed during a garbage collection cycle. When an object is reclaimed, its internal reference fields need to be examined to determine whether any objects that it references have become garbage.

The main advantage of reference counting is its inherent incrementality. The process of updating references is done while the mutator runs. More importantly, reclamation can be closely interleaved with mutator execution. Reference counting can be effectively used to provide low average pause times.

However, there are three major problems with reference counting:

- **Storage Overhead.** Storing the reference count requires up to one word of space overhead per object.

- **Cyclic References.** Reference counting fails to collect garbage cycles. In the most simplistic scenario, if two garbage objects reference each other, but they are not referenced by any other object, their reference counts will never drop to zero, and they will never be collected.
- **Poor Efficiency.** Since reference counts must be updated for every reference that is created or deleted by the application, the detection phase of reference counting results in an overhead that is proportional to mutator activity. This problem is exacerbated for stack variables, since they are short-lived and exist only during the lifetime of the activation. To alleviate this problem somewhat, deferred reference counting [28] is used. Objects in the stack are not reference counted, and only heap-to-heap references are tracked. But garbage collection requires that stack objects be part of the root set. Consequently, stack objects are scanned for references to heap objects once in a while (usually at short intervals).

### 2.3.2 Mark-Sweep Collection

In mark-sweep collection, the two phases of garbage collection, detection and reclamation, can be identified quite distinctly:

- **Mark phase.** Objects that are directly or transitively reachable from the root set are marked live. Marking can be done by setting a bit in the object header or in a global bitmap.
- **Sweep phase.** In this phase, objects that are not marked in the mark phase are swept or reclaimed for use by the mutator. Freed objects are linked into a free-list maintained by the heap allocator. Allocating from a free-list, however, is more expensive than allocating from a contiguous memory region.

The mark-sweep technique handles cyclic references automatically, and it imposes no overhead during object manipulation. However, some amount of work is required at object creation time, to initialize the object header (if the mark bit is stored in the header). There are three more important problems with mark-sweep collection.

- **Fragmentation.** Since reclamation is done in place, in time, free areas are interspersed with live objects, leading to a fragmented heap. Object allocation might fail even though the total amount of free space is sufficient to honor the allocation request. Various techniques are used to mitigate this problem, e.g., maintaining free lists with different block sizes called segregated lists, and buddy lists in which blocks from adjacent lists are coalesced [64].
- **Collection cost.** The collection cost of mark-sweep collection is proportional to the size of the heap. While reclaiming objects during the sweep phase, garbage as well as live objects are visited.
- **Poor Locality of Reference.** Since objects are allocated and freed in place, freshly created objects may be spatially closer to older objects, leading to poor locality.

### 2.3.3 Mark-Compact Collection

To handle fragmentation, poor locality, and expensive allocation associated with mark-sweep collection, mark-compact collection was invented [27]. In mark-compact collection, the initial marking phase is similar to the marking phase of mark-sweep collection. However, the reclamation phase attempts to compact

live data into one contiguous region of the heap. This solves the fragmentation problem. As a result of compacting live data, free space exists as a contiguous region. Consequently, allocation is inexpensive as it involves only an pointer increment into the contiguous free space. Locality is also improved, since objects of a similar age are clustered together in space.

However, the actual process of compaction can be quite expensive, and usually requires at least two passes: one pass to identify the new locations for the live objects, and another to update pointers and actually move the objects.

### 2.3.4 Copying Garbage Collection

Compaction is an inherent part of copying garbage collection, in which all live data is copied to one part of the heap, so that it is contiguously laid out. The rest of the heap is then considered to be free, and can be used by the allocator for future allocations. Copying collection is often considered to be *implicit*, or scavenging, since garbage is not explicitly identified and reclaimed.

The heap is usually divided into two equal parts or *semispaces*: the *from space* and the *to space*. All allocation is from the from space, which is the considered to be the “current” semispace. The to space is always empty while the application executes. During a collection, live data is copied from the from space to the to space. The from space now contains only garbage and can be reclaimed. This is done by swapping the roles of the two spaces (usually by merely updating references to the from space and the to space). Thus, at the end of the collection cycle, the to space is empty and live data has been contiguously arranged in the from space.

Cheney’s traversal algorithm [25] is the most popular method for identifying live data. As mentioned before, the *root set* for a garbage collection consists of static variables, stack variables, and register references. The objects in this set are first added to a queue, which is then scanned in a breadth-first manner to identify objects that are reachable from the root set. These objects are in turn added to the queue, in order to identify heap objects reachable from them, and this recursive process continues until all live objects have been traced. Every object that is identified as live is copied to the to space, and a *forwarding bit*, is set in the object’s header. Along with the forwarding bit, a forwarding pointer is also stored, which indicates the new location of the object. The forwarding pointer enables updates to all pointers that refer to the object. The use of the forwarding bit avoids duplicate copying of live objects and guarantees termination.

Copying collection has several advantages. Similar to the mark-compact technique, it has the fastest allocation mechanism, called the *bump-pointer*, which can be implemented as a simple increment into the free space (usually, along with a boundary check, to decide whether a garbage collection should be initiated). Fragmentation is non-existent since the algorithm is inherently compacting. Locality, however, may not always be improved – Cheney’s breadth-first traversal may not preserve locality of reference. Depth-first traversal has been used in some recent implementations [61].

Unlike mark-sweep and its variation, the mark-compact algorithm, the amount of work during copying garbage collection is proportional to the amount of live data, and not to the size of the heap. In addition, unlike mark-compact collection, a single pass over the live data is sufficient. Mutator overhead is similar to mark-sweep collection, since during object creation, the forwarding pointer and the forwarding bit have to be initialized.

The biggest disadvantage of semispace copying collection is that the heap area available to an application is only half of the entire heap space. An application’s memory requirement is dou-

bled compared to mark-sweep or mark-compact collection. Another major disadvantage is that the process of reclamation is quite slow, given the need to copy every single live object. However, since the bump pointer allocation is a feature of copying collection, this algorithm performs quite well when garbage collection cycles are infrequent.

### 2.3.5 Generational Garbage Collection

As noted previously, mark-sweep collection needs to scan the entire heap space, a process which can be quite expensive. Mark-compact collection performs a number of passes over the entire heap, and if the amount of live data is high, performance suffers. Copying collection also has high overhead, particularly if the amount of live data is proportional to the size of a semispace.

Live data, however, has one interesting property, which is known as the *weak generational hypothesis* [47, 62]. It states that most dynamically allocated objects (between 80 to 90%) have very short lifetimes, and only a small percentage of objects live much longer. Generational garbage collection exploits this property. Objects that have been recently created (also called *young* objects), are segregated into a *nursery* (young object) area. As objects age in the heap (usually indicated by their survival after one or more garbage collection cycles), they are promoted (copied) to another area of the heap, called the *mature space*. The promoted objects are called *old* or *mature* objects.

The basic idea behind this approach is that the garbage collector should not have to process (either mark, or copy) objects that are going to survive well into the application’s lifetime. Effort can be concentrated on collecting young objects, which will die sooner. The process of collecting the nursery area is called a *minor collection*. The mature space should also be examined once in a while to check whether older objects have become garbage. This is called a *major collection*, and is done less frequently. New objects are allocated only from the nursery. Any collection algorithm could be used to collect the nursery and the mature space, but promotions always *copy* live data from the nursery.

The generational technique leads to reduced garbage collection overhead, since excessive processing of live data is eliminated. This comes at a price – to perform minor collections without collecting the mature space, some book-keeping is required. It is necessary to keep track of references from the mature space to the nursery, otherwise objects in the nursery that are referenced by objects in the mature space will be incorrectly reclaimed as garbage. This is done by means of a *write barrier*, which is a conditional or an unconditional check inserted into the compiled code in order to track pointer stores. The objects in the mature space that reference nursery objects must be remembered and included in the root set of objects for minor collection. We will discuss various write barrier implementations and their relative performance in Section 3.1.2.

There have been implementations of generational collection with multiple generations [23, 41] instead of only two. But it has been argued that multiple generations may not lead to much improvement in performance [3, 13]. The generational technique has been regarded as most effective in practice, and it has been successfully combined with all basic collection mechanisms [3, 66, 15]. However, given its book-keeping overhead, incorporating generations into a garbage collector is not always beneficial. In particular, applications which have a large number of stores from the mature space to the nursery, may perform poorly with generational collection, e.g., a long-lived array which references many newly created nursery objects.

### 3. MODERN GARBAGE COLLECTION TECHNIQUES

In this section, we will examine modern advances in garbage collection.

The basic garbage collection algorithms discussed in the previous section are not suitable for modern virtual machines. Modern garbage collection techniques are aimed at enabling efficient application execution in virtual machines. They can be classified into two types: techniques for maximizing application throughput, and techniques for minimizing application pause time. We will also examine some supporting techniques that enhance garbage collector performance in this section. In addition, we will discuss some garbage collector performance evaluation studies.

#### 3.1 Maximizing Throughput

In this section we will examine modern garbage collection techniques that aim at maximizing application throughput. This entails minimizing garbage collection overhead and reducing the frequency of garbage collections.

##### 3.1.1 Generational Copying Collection

Appel's seminal work on generational copying collector [3], although dated (1989), is still considered to describe a very effective garbage collection algorithm. It is used in many modern virtual machines [2, 61] and even very recent garbage collector techniques [53, 13, 59] have been compared to it.

In Appel's collector, objects are separated into young and old generations, according to the generational principle. Appel noted that for many programming languages (Lisp, ML, Prolog) of the time, it was rare to have cycles between young and old objects. Once objects were created and initialized, assignments were rarely made to them. This would imply that there are very few pointers from old objects to young objects. In modern programming languages, this principle seems to be just as valid. To keep track of the few old to young references, Appel considered the use of three techniques – assignment tables [47], special hardware to examine pointer stores [49], and maintaining a list of old objects that point to young objects [62]. He preferred maintaining a list of old objects, since it does not require special hardware and also does not use slow indirect addressing unlike assignment tables.

Appel made use of Cheney's [25] copying garbage collection for both the nursery as well as the mature space. Appel also proposed a fast allocator that involves a check against the free-space limit to determine whether a collection should be initiated, followed by a single arithmetic operation on the free space pointer. This allocator (modified slightly for non-functional programming languages) is the fastest known, even today. All allocation is from the nursery.

Appel recommended the use of only two generations, since this would allow the nursery to be sufficiently large so as to minimize promotions. Appel did not recommend the use of a fixed size nursery, unlike the one used in some other implementations [62]. Instead, the nursery initially occupies half the heap, and the mature space is empty. As minor collections occur, the nursery shrinks and the mature space grows. After a major collection, the mature space contains live old data, and the nursery automatically expands. Thus, the boundary between the nursery and the mature space varies dynamically. This organization enables much better performance compared to a fixed-size young object space [13].

Appel further recommended that the old object space be laid out in such a manner that a failure to allocate from the old space would generate a segmentation fault, which could be handled and a major collection be initiated. In addition, he proposed a heuristic

to determine when to request more memory from the operating system. He noted that an application will never run out of space if the amount of live data is less than half the size of the heap. However, he pointed out that if the amount of live data approaches half the size of the heap, performance will suffer significantly. He therefore recommended that the runtime should request more memory from the operating system, if the ratio of heap size to the amount of live data is less than three (i.e. live data occupies one-third of the heap). Appel implemented his algorithm in Standard ML of New Jersey, and demonstrated that even for applications with a very high allocation rate, the garbage collection overhead was only 5 to 10%.

Generational garbage collectors keep track of references from the mature space to the nursery, using write barriers.

##### 3.1.2 Write Barriers

Before we look at the next modern garbage collection technique, let us discuss write barriers in some detail.

References from old to young objects need to be remembered, so that a minor collection may occur independently of a major collection. Old objects that reference young objects are included in the root set for minor collection. Write barriers are used to keep track of pointer stores. Another use for write barriers is in incremental collections, where the process of reclamation is performed incrementally across collections. Incremental schemes divide the heap into fixed sized areas (increments) that are collected independently. Write barriers are required to ensure that cross-increment references are remembered. Otherwise, some objects in the increment that is currently being collected, might be incorrectly considered to be garbage and reclaimed.

There are different write barrier techniques that differ mainly in the granularity of information stored. Hosking et al [37] compared these three techniques in a virtual machine with a Smalltalk interpreter. The three mechanisms they considered are: *remembered sets*, *card marking*, and *page protection*.

Remembered sets are the most accurate of the three, since they record the actual old object (or the slot containing the object) that references a young object. Hosking et al implemented remembered sets using two alternative implementations. A hash table is the most standard way of storing remembered set entries, however, insertion overhead might be considerable. Consequently, they introduced the concept of a *sequential store buffer* (SSB), which consists of pages arranged contiguously and bounded by a limit or guard page. This allows the use of a simple pointer increment-and-store operation to remember entries. An attempt to store into the SSB past its limit is trapped by an operating system trap, which can be handled by the runtime.

Another scheme for remembering cross-generational stores is card marking, in which the heap is divided into multiple cards, with every card represented by a unique entry in a *card table*. The source card that contains the old object is marked, instead of remembering the object itself. Thus the level of granularity is much coarser. This implies that the pointer store check can be performed quickly, but at the cost of garbage collection time overhead, since the entire card has to be scanned to locate all references to nursery objects. The card marking scheme as originally introduced by Wilson [65] made use of a *bit* per card. The authors used a *byte* per card, which makes the process of checking and marking more efficient, since the smallest unit of memory access on most architectures is a byte. In a related work, Chambers et al [24] also used a byte marking scheme.

Page protection further extends the concept of sacrificing finer granularity in exchange for less mutator overhead. Instead of

checking pointer stores at runtime, page protection uses the paging hardware’s ability to trap writes to protected pages, i.e., the level of granularity is a page. During a minor garbage collection cycle, the entire page has to be scanned for old-to-young references.

Hosking et al tested application performance with the three mechanisms for five Smalltalk applications. Since the runtime used an interpreter, the difference in application execution time for the three different schemes was not large enough to be visible as a significant percentage. Even so, the authors observed that remembered sets provide the best performance, since there is no scanning overhead during minor collection. The SSB is more efficient than hash table insertion when the pointer storage rate is high. However, trapping and handling the overflow condition for the SSB has some negative impact on application performance. Card marking with a card-size of 256 or 512 bytes also seems fairly efficient. Surprisingly, page protection seems to suffer not as a result of overhead due to trapping and handling protection faults, but mainly due to the coarse granularity of information stored, which results in slower garbage collection performance.

Although it should be very interesting to see how these schemes perform when application code is compiled (instead of it being interpreted), to our knowledge, there is no study that directly compares performance of the three mechanisms in a runtime with an optimizing compiler.

Hoelzle [35] noted that the pointer store check in Wilson’s basic card marking scheme [65] is quite slow, since a bit vector needs to be read from memory, updated, and then written back. Chambers et al [24] tried to improve on this by using a byte per card, instead of a single bit. On most architectures, marking a byte is much faster than marking a bit – on a SPARC, this process can be performed in 3 instructions. Hoelzle, attempted to further reduce the pointer store check overhead, by reducing the three-instruction write barrier to a two-instruction write barrier. This is significant, since he also demonstrated that pointer store checks constitute about half of the performance overhead associated with card marking (the other half is the time taken to scan the card table during minor garbage collection).

Hoelzle used a relaxed card marking scheme that uses an approximation during the card marking process, at the cost of some additional overhead during minor garbage collection. According to this scheme, a card containing an old-to-young reference is not remembered precisely, but rather, an entry in the card table corresponds to more than one card. This approximation saves one instruction per pointer store, compared to Chambers et al’s accurate card marking. However, the extra scanning overhead due to this approximation might be too large for large objects and arrays. For such objects, Hoelzle used accurate card marking. The overhead due to store checks is determined by running benchmark programs with an instruction-level simulator, which models the exact hardware behavior, including cache behavior. The store check overhead accounts for 0.5 to 8% of the total application execution time. This is lower than the overhead that Chamber et al reported (3 to 24%).

Hoelzle also compared his technique against Hosking et al’s sequential store buffer (SSB) technique. He pointed out that for the SSB, the overhead during collection is proportional to the number of pointer stores performed between garbage collections. For card marking, overhead during collection is proportional to the live area of the old object space. If the number of pointer stores are large, then the SSB can become quite large. This is especially so for the benchmark programs considered by Hoelzle. These are SELF programs for which pointer stores are quite frequent. Con-

sequently, the estimated overhead of SSB is larger than using card marking (Hoelzle did not measure the actual overhead, but rather estimated it by considering what he believed was a “realistic” SSB implementation).

Hoelzle’s study is however, only relevant to card marking. Most modern garbage collectors [13, 59, 38] make use of remembered sets, which has a comparatively low scavenge time overhead. Remembered sets require a conditional check during pointer stores to determine whether the source and the destination object, reside in the mature space and the nursery, respectively. The mutator overhead can be reduced to a large extent by using partial inlining of the write barrier pointer store check as described by Blackburn et al [14]. They measured the mutator overhead for three different ways of implementing the check – a fully inlined write barrier, a partially inlined write barrier, and an out-of-line write barrier. They used an implementation of Appel’s generational copying collector for their measurements in the Jikes RVM [2, 1]. The authors also compared their results against an implementation in a non-generational collector, which does not require a write barrier. In addition, the authors attempted to evaluate whether it is beneficial to remember the old *object* that references the young object, or the *slot* (pointer field) that contains the old object. In either case, the object or slot is remembered using a variation of Hudson et al’s sequential store buffer (SSB) technique, with a power-of-two aligned buffer.

Blackburn et al divided the pointer store check into two parts – a *fast path* which performs the check to determine whether the reference is from an old to a young object, and a *slow path*, which actually inserts the object (or its slot) into the SSB. The fast paths for the two cases, viz., remembering the old object or remembering the slot, are different. For the former (remembering the object), the old object’s header word is checked for the presence of an *OBJECT\_BARRIER* bit. If the bit has not been set, the slow path is taken, which will set the bit as part of the process of remembering the object. The *OBJECT\_BARRIER* bit is cleared for every object when it is first created. Since, new objects are only allocated from the young space, correctness is ensured. For the case in which the *slot* containing the object is to be remembered, the fast path is implemented using a technique by Stefanovic et al [60]. The young object space is located in high memory and the old object space is in lower memory, with both spaces aligned on a boundary ( $2^k$ ). Consequently, a simple bit-mask-and-shift can be used instead of an and operation.

A fully inlined write barrier implies that the fast, as well as slow paths are inlined at the site of the pointer store. A partially inlined check inlines only the fast path, and a out-of-line check makes use of a direct functional call without any inlining. The authors measured the compilation time for the three implementations by fully compiling their benchmark programs using the Jikes RVM’s optimizing compiler [22]. They also measured application performance without considering compilation time. The compilation measurements show that full inlining incurs a heavy compilation time penalty (up to 25% worse compared to partial inlining, and 38% worse compared to the out-of-line check). The out-of-line check and store has the least compilation time overhead. The authors then showed that the slow path is rarely taken for their set of benchmark programs (0.15 to 3%). This implies that the fully inlined write barrier will not improve application performance by a noticeable amount. In fact, the authors reported that full inlining actually *degrades* application performance. This is probably due to poor locality and register allocator performance, as a consequence of excess code generated at pointer store sites. The out-of-line write barrier performs worse, since an out-of-line function

has to be invoked for every pointer store. Partial inlining enables the best application performance. The authors also showed that the partially inlined slot barrier (in which the remembered set holds slots that contain the old object) performs better than the partially inlined object barrier (the remembered set holds the actual old objects). This is probably due to the fact that for the object barrier, the collector must scan the stored old object for pointers from the mature space to the nursery. The slot barrier remembers more pointers and a scan of the old object is not necessary.

### 3.1.3 Generational Mark-Copy

Sachindran et al [53] described a generational algorithm that uses copying collection for the young object space (nursery), and a mark-copy collection technique for the older generation. This algorithm is designed to provide good throughput, while decreasing the amount of memory required. Generational mark-copy can also be used for incremental collection (the authors discussed but did not implement an incremental collector).

We will only discuss mature space collection (major collection) here, since a minor collection is a copying collection similar to Appel's technique. Major collection makes use of marking along with copying. The heap space is divided into fixed-size windows, which are labeled sequentially. During a mature collection, windows are arranged into groups with one or more adjacent windows in one group. Groups containing lower numbered windows are collected before groups containing higher numbered windows. To keep track of references from a higher numbered window to a lower numbered window, per-window remembered sets are maintained. Since a lower numbered window will always be collected before a higher numbered window, it is not necessary to keep track of references from a lower numbered window to a higher numbered window. The first phase of a major collection is a marking phase, in which per-window remembered sets are populated, and the volume of live data per window is calculated.

The amount of per-window live data is used to decide how to arrange windows into groups. During the second phase of collection, i.e., copying, the live data in a group of windows is copied into windows in a higher part of the heap. Copying occurs in multiple passes with one group being collected per pass. In other words, major collection needs multiple copying passes to complete, but in each pass, only a limited amount of live data is copied. This is unlike the Appel collector, which copies all live data in the old object space in one go. The result is a better use of the available heap space. The major advantage of this algorithm is that the copy reserve area can be quite small compared to Appel's generational copying collector, in which live data can never exceed 50% of the total heap space.

Blackburn et al discussed a version of the collector that does not require a remembered set, but instead stores an object's logical location in an extra per-object header word. Reference fields inside the object are replaced with the logical location of the referents, during the marking phase. During copying, the true object locations are calculated and the reference fields are updated. Objects are copied to the actual location calculated from the logical location stored in its header.

The generational mark-copy algorithm is able to run applications in a heap size that is 1.12 to 1.25 times the maximum live data size of an application. Moreover, this algorithm improves application performance by 5-10% in moderate size heaps, over Appel's generational copying collection. This is attributed to the reduction in the data copied during a full heap collection. Full heap collections are however, 20% slower compared to an Appel-style collector. This is due to the fact that multiple copy passes

are required during a mark-copy major collection, especially if the amount of live data is large. However, the generational mark-copy collector performs full heap collections less frequently. The reason for this is that generational mark-copy can defer full heap collection until the heap is significantly occupied, as opposed to the Appel collector, in which a full heap collection needs to be performed when the heap is half full.

## 3.2 Minimizing Pause Time

Minimizing the amount of time that an application needs to pause while garbage collection is in progress increases responsiveness. The basic idea is to not collect the entire heap in one garbage collection cycle, but rather, to collect a fixed area of the heap per garbage collection cycle, i.e., *incrementally*. As stated earlier, this is called incremental garbage collection. However, an important drawback of incremental collection is that it is not complete, i.e., it does not guarantee that all garbage will be collected in *one* garbage collection cycle. It does guarantee that garbage will be collected *eventually*, including cyclic garbage. Since a fixed amount of work is done per garbage collection cycle, the GC cycle completes faster, but GC may have to be triggered more frequently. Consequently, incremental collection commonly does not enable high application throughput.

### 3.2.1 The Train Algorithm

Hudson and Moss [38] described an incremental, compacting algorithm for the collection of the mature (old object) space in generational collection. The incremental nature of the algorithm prevents disruptive pauses during application execution. The authors used copying collection to enable compaction of the mature space. The mature space collector for the HotSpot virtual machine [61] is based on this algorithm.

Hudson et al's algorithm is implemented in a garbage collection toolkit, that the authors developed in a prior work [39]. The heap is divided into blocks, where the typical block size is 64KB. The mature object space is divided into areas or *cars*, with one or more cars constituting a *train*. The size of an area is limited, and as such, the amount of work during each collection is bounded.

Cars are numbered in a linear sequential manner within a train, and they are scavenged in that order. Trains are also sequentially marked and processed in a round-robin manner. Every car stores a *remembered set* of objects in high-numbered cars that reference objects in that car. A train is also associated with a remembered set, which is the union of the objects in the remembered set of its cars, minus any intra-train references.

Only one car is collected per garbage collection cycle. Since cars within a train are collected sequentially, the remembered set for a car does not need to keep track of references from objects in that car to objects in a higher numbered car. A lower numbered car will always be collected before a higher numbered car.

A major collection proceeds as follows. Trains are scavenged in a round-robin manner, based on their sequence numbers. In the first step of the algorithm, the current train is checked to determine whether its remembered set is empty. If so, then the entire train can be freed. Otherwise, the cars in a train are processed in a sequentially manner. Any object that needs to be copied, is copied into a train which has a car that is capable of accommodating the object. Otherwise, a new train is created and the object is copied to the first car of the new train. The car that is currently being scavenged is called the *from car*. First, any objects that are referenced by the roots are copied. Using Cheney's breadth-first traversal, all objects reachable from these are then copied. Also, at this time, live objects are promoted (copied) from the

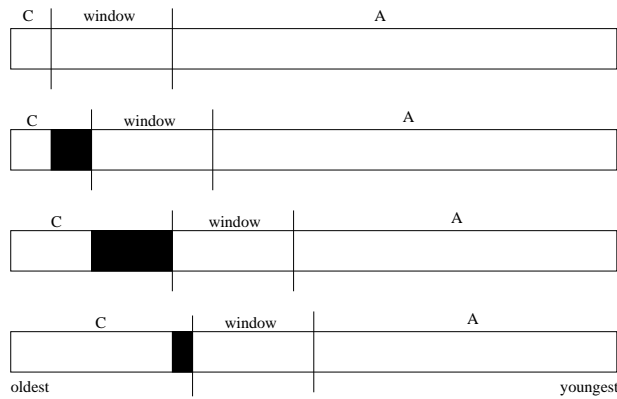


Figure 1: Example of Older-First Collection

young object space. Next, the remembered set for the from car is scanned and all objects referenced by objects in other trains (source objects), are copied to a car in the train that the source object belongs to. Objects that are referenced by objects in the same train are copied to the last car in the *same* train.

### 3.2.2 Older-First Garbage Collection

Stefanovic et al made an interesting argument against generational garbage collection’s propensity towards concentrating most effort on young objects [60]. Their work presented an algorithm for older-first garbage collection, and a simulation to study its benefit. Recently, they presented an implementation of older-first garbage collection [59] in the Jikes RVM. We next discuss this more recent work.

Stefanovic et al argued that generational collectors spend significant effort on objects that have been created very recently, without allowing them enough time to die. Even though the weak generational hypothesis holds in principle, generational collectors collect the youngest generation too soon. The authors proposed a collection technique that collects objects in a *fixed-sized window*, as it slides from the *oldest* area of the heap, towards younger objects. Figure 1 shows an example of how this algorithm works. A fixed-sized window “slides” from the older part of the heap towards younger objects, from the left to the right. A collection of a window is initiated when the heap is full. The survivors of a collection are copied to the *copy area* (marked C), and the window is advanced over surviving objects (dark region), towards the allocation area (marked A). New objects are allocated from the allocation area. The window only slides by an amount equal to the size of the surviving data after a collection. When the window reaches the right-most end, i.e., area A is empty, the two areas (A and C) are flipped. Abstractly, this means that the window is reset to the older-most part of the heap.

Stefanovic et al implemented the sliding window technique by dividing the address space into *frames*, where the size of a frame is equal to the size of a window (which is fixed). The older first collector needs to remember pointers from older frames to the frame being currently collected (the window). Each frame is assigned a *time of death* (TOD) value, in order to impose an ordering on frames. Frames are further divided into blocks, which are mapped and unmapped on demand. New objects are allocated from one or more blocks, depending on the size of the object. The allocation (A) and the copy (C) regions are assigned increasing TOD values for every frame that they request. The starting TOD value for the C region is established to be suffi-

ciently large, so as to avoid collisions with TOD values for the A region. The relative age of frames is defined by their respective TOD values. Write barriers are necessary to record pointers from older to younger frames. The authors noted that stores need to be remembered only if the TOD value of the target block’s frame is less than the TOD value of the source block’s frame. Intra-frame references are never recorded. This type of write barrier is more expensive than an address-ordered write barrier which is usually used in generational collectors. With an address-ordered write barrier, old and young objects can be identified by checking object addresses. However, in Stefanovic et al’s implementation, old and young frames cannot be determined by examining object addresses.

Stefanovic et al compared their collector against a generational copying collector with a fixed-size nursery, and Appel’s collectors. For their comparison, they used three metrics – the *mark/cons ratio*, application pause-time, and application execution time. The mark/cons ratio deserves special mention. It can be defined as the ratio of the total amount of data copied by the collector to the total amount of data freshly allocated by the program. Mark/cons ratio is considered to be a good indicator of performance for copying collectors, since the copying of live data is the major part of copying collection. Larger the mark/cons ratio, larger is the performance overhead due to copying, and consequently, garbage collection. The authors showed that the mark/cons ratio is reduced overall (across applications and heap sizes) by using their older-first collector, since unnecessary copying of very young objects is avoided. The older-first collector is also able to reduce pause times for many benchmarks (but not for all benchmarks). The total application execution time also seems to be reduced for 7 out of 10 benchmarks presented. However, the book-keeping overhead for some benchmarks is rather large with older first collection. A greater number of pointers are recorded across write barriers, and more processing is done while scanning the remembered set during garbage collection time. An inability to use an address-ordered write barrier is also a major drawback. Even so, the overall performance shows improvement compared to both versions of the generational copying collector, since the mark/cons ratio is significantly reduced.

### 3.2.3 Ulterior Reference Counting

Blackburn et al [15] combined deferred reference counting [28] with the generational principle, to implement an algorithm that uses copying collection for the nursery (minor collection) and reference counting for the mature space. The purpose was to combine the good performance of generational collection with the low pause times and responsiveness of reference counting. The authors used a bounded nursery, along with reference counting for the old object space, hence the algorithm is called bounded generational reference counting (BG-RC). A bounded nursery enables better performance than a fixed size nursery, but exhibits worse performance compared to a variable sized nursery. The authors used a bounded nursery in order to bound the amount of work performed during a minor collection, thus improving pause time.

Blackburn et al integrated various optimizations into their BG-RC collector, viz., *deferral* [28], *buffering* of increments and decrements [28, 7], and *coalescing* [46]. As mentioned in Section 2.3.1, deferral involves examining heavily mutated pointers only periodically. In the BG-RC collector, reference counting for young objects, stack variables, and registers are deferred. This enables incrementality at the price of efficiency. Increments and decrements are buffered to avoid applying them too frequently. Coalescing makes use of the observation that only the initial and the

final values of pointer fields of objects are relevant, i.e., intermediate mutations can be ignored. The basic reference counting algorithm cannot handle cycles. The authors used Bacon et al's trial deletion algorithm [9] to detect and reclaim garbage cycles.

Blackburn et al compared their BC-RC collector to a pure reference counting collector, a pure mark-sweep collector, and a bounded generational mark-sweep collector (BG-MS). They observed that BG-RC improves application performance by around 2% on average (compared to BG-MS). More interestingly, the maximum pause time is decreased by a factor of 4 on average. In tight heaps, however, BG-RC's performance degrades more rapidly than BG-MS. This is probably because BG-RC is unable to quickly reclaim cyclic garbage.

The authors also claimed that their collector was probably superior to Azatchi et al's [6] generational reference counting algorithm, which uses concurrent non-moving collection with free-list allocation for both generations. Azatchi et al's collector enumerates all young objects and stores them in a list, unlike Blackburn et al's collector, which only needs to enumerate young objects that are live. The former, consequently has a higher memory and garbage collection time overhead.

### 3.2.4 Concurrent Garbage Collection

Most of the garbage collection techniques discussed in this paper do not scale well to large servers with multiple processors, and multi-gigabyte heaps. There are millions of client requests for such systems, with clients expecting low response times. Garbage collection for large servers needs to be designed for high throughput, low pause times, and good scaling on multiple processors. Concurrent garbage collection has been designed to satisfy this requirement. A concurrent garbage collector runs simultaneously with the mutator. The mutator only has to be paused for a short period of time (the reason for this is explained below). This type of collector is therefore called a *mostly-concurrent* collector [17, 51, 11]. The work by Barabash et al [11], subsumes Boehm's original mostly-concurrent algorithm [17], which we discuss only briefly.

Barabash et al improved upon Boehm's original work on mostly-concurrent collection, by improving throughput, reducing the memory footprint, and improving cache behavior. We next briefly describe Boehm's collector.

In Boehm's collector, the garbage collector runs as a separate thread (usually on a different processor), simultaneously, but asynchronously, with the mutator. The collector marks and scans the heap *while* the mutator is modifying objects. This requires some co-operation between the mutator and the collector to ensure correctness and safety. Consider a scenario in which an object O has been marked live by the collector. If the mutator now modifies O such that it references an object A, which has not been traced by the collector, object A may be incorrectly reclaimed by the collector. The problem is that the object graph may be modified by the mutator while marking and reclamation is in progress.

In order ensure correctness, a card marking scheme is used. The collector first clears the dirty bits of all cards. Then the collector starts its marking phase to identify live objects. While the collector executes, for every object that the mutator modifies, the mutator marks the corresponding card (the card that contains the object) as dirty. This is similar to a conventional card marking write barrier. After the collector finishes its marking and tracing phase, it scans all dirty cards, clearing their dirty bits, and retracing through all marked objects on the cards. This ensures that the collector maintains a correct view of the object graph which may have changed due to mutator activity.

Two card cleaning phases are run: one that executes concurrently with the mutator, and another in which the mutator is *stopped* for a short while. The second phase is necessary to ensure completion of the card cleaning process. The reduction in pause time over a conventional stop-the-world collector is based on the fact that marking can be done concurrently, and less work is performed during the second, potentially disruptive card cleaning phase. The reclamation phase also runs concurrently with the mutator.

Barabash et al's first improvement to Boehm's collector is based on the reduction of tracing work that is done per concurrent garbage collection cycle. In the original Boehm's collector, many objects are scanned twice – once during the marking and tracing pass of the collector, and a second time during the card cleaning phase. The authors noted that during the marking phase, it is not necessary to scan an object on a dirty card, since it will later be scanned during the card cleaning phase.

The second improvement results from the reduction in the number of dirty cards. By reducing the number of dirty cards, pause time will improve since less number of cards will have to be scanned in the second, potentially disruptive card cleaning phase. Upon mutation of an object, the corresponding card needs to be marked only if the object has already been marked during the tracing phase of collection. However, implementing this check may degrade performance due to increased write barrier overhead. Instead, the authors proposed two alternative techniques that capture this idea.

In the first technique, the card is marked dirty as usual, however, a procedure runs once in a while that unmarks the card if it contains an object that has been traced. This requires the use of a second card table that maintains information about traced objects.

The second technique for undirtying cards uses allocation caches, which are processor local chunks of memory from which allocation can be performed without synchronization [19, 43]. When a processor has exhausted its allocation cache, it requests another cache. The authors stated that this is a good time to undirty all cards contained in the allocation cache.

The use of an allocation cache also avoids maintaining the second card table as in the first technique, since a bit can be maintained per object to indicate whether the object belongs to an active allocation cache. If the bit is set, then the collector will refrain from tracing the object immediately.

Barabash et al compared their improved collector to an implementation of Boehm's original collector on a 4 processor and a 6 processor machine for the SPECjbb2000 [56] and the SpecJVM98 [57] benchmarks. A reduction in the tracing work increases the amount of CPU time available to the mutator, which improves application and cache performance. Cache coherency misses are also reduced as a result of the lower tracing rate. This is due to the reduced possibility of read/write contention between the mutator and the collector as they try to access the same object.

The authors further reported a lower heap residency than Boehm's collector. This is probably because of reduction in the amount of *floating garbage* due to the deferred tracing mechanism. Floating garbage is unreclaimed garbage resulting from objects that were marked live during the initial marking phase, but then became unreachable due to mutator activity.

The authors reported an application performance improvement of about 26%, a reduction in heap usage by about 13%, and a 2 to 6% reduction in cache miss rate, with their improved collector. They also reported no substantial change in pause time. This collector is now a part of IBM's Java Virtual Machine 1.4.0.

Printezis et al [51] combined mostly concurrent collection with the generational collection technique. Their implementation of

the generational collection algorithm involves copying collection for the nursery, and mark-sweep reclamation of the mature space. To support concurrency, a stop-the-world traversal and marking which scans the entire object graph cannot be used. Instead, they used an external marking bitmap, which is used to mark objects that are reachable *directly* from the roots during a stop-the-world root scan phase. The entire heap is not traced in this phase. The concurrent marking phase makes use of this bit map to perform a linear traversal and marking of the rest of the heap space.

The use of card marking by both generational collection and mostly-concurrent collection, in the same system, requires some special consideration. The use of card marking by the two algorithms is different. The mostly concurrent algorithm tracks needs to track object mutations that have occurred since the start of the current concurrent marking phase. Generational collection uses card marking to keep track of old to young references. During a minor collection, cards are scanned for old to young references, and if none are found, then the card can be unmarked (or marked clean). However, the mostly-concurrent collector needs to be informed about the dirty status of the card. This is done via a data structure called the *mod union table*. This is a bit vector which represents the union of sets of cards modified between each of the minor collections that occur during the concurrent marking phase.

Printezis et al discussed some further optimizations, like support for a *linear allocation mode*, so that promotion of small (not all) objects can be handled by fast linear allocation from free chunks. They also implemented a technique called *concurrent precleaning*, which concurrently scans dirty cards before the final stop-the-world scanning phase. This is similar to Barabash et al's two phase card scanning technique, in which the first phase is a concurrent scanning pass. Another improvement is that the sweeping phase coalesces smaller consecutive free chunks into larger chunks.

Printezis et al measured application performance with their collector using a synthetic benchmark. They observed that using mostly-concurrent collection significantly decreases old-generation pause times. Use of the mostly-concurrent collector has a higher heap residency, compared to non-concurrent generational collection. Overall application execution times are relatively unchanged, with mostly-concurrent collection being a little faster. However, the synthetic benchmark does not have a very high mutation rate.

The authors also validated their findings for some of the SpecJVM98 benchmarks. As with the synthetic benchmark, they found similar application execution times, reduced paused times, and higher heap residency, for their mostly-concurrent generational collector compared to the generational collector. The authors argued that for applications with low promotion rates, mostly-concurrent generational collection improves pause time. Minor collection is slower for the mostly-concurrent generational collector, since promotion does not always make use of fast linear allocation. The original generational collector that the mostly-concurrent generational collection is being compared against uses mark-compact collection for its mature space. Concurrent compaction is expensive and infeasible, and hence, the mostly-concurrent generational collector uses mark-sweep collection for the older generation. Even so, this algorithm would perform well on a multiple processor system, if the collector runs on a separate processor.

### 3.3 Supporting Techniques

In this section we examine some techniques that are not garbage collection algorithms in themselves, rather, they are intended to be

used with other garbage collection techniques to reduce garbage collection overhead and to improve application performance.

#### 3.3.1 Large Object Spaces

Apart from object lifetime, object size also seems to be a factor for segregation of objects into separate heap spaces, as studied by Hicks et al [34]. Garbage collection overhead for objects above a certain size threshold, called large objects, degrades garbage collector performance due to high *per-byte* costs. Per-byte costs include copying overhead and the time spent in tracing references inside a large object. Hicks et al concluded that it is beneficial to segregate large objects into a separate region of the heap, called a *large object space* (LOS), and collect this region using a non-copying collector, like mark-sweep.

Marking an object during mark-sweep collection, or setting the forwarding pointer during copying collection are *per-object* costs. The larger the size of the object, the greater is the overhead for copying it or tracing internal pointers. For objects with no internal pointers, non-copying collection is inherently beneficial, since it involves no per-byte costs. Consequently, the authors argued for segregating objects into a large object space which is collected using a non-copying collector. In addition, and perhaps surprisingly, the authors showed that adding pointer-containing objects to the large object space also seems to have some benefit. The authors also suggested two different schemes for allocating objects into the LOS. Objects can be allocated directly into the LOS by adding a check during allocation time, or they can be promoted the LOS after initial allocation. However, the latter technique might prove to be more expensive due to copying overhead. The authors also evaluated two different non-copying algorithms for the LOS – Mark-sweep and Baker's treadmill collector [10].

Hicks et al performed their study in a testbed, that enables replay of snapshots captured during an actual execution run of an application. A snapshot of the heap is captured in a format that is understood by the replay program. The free list management and allocation overhead for a non-copying mark-sweep collector is amortized over the size of the object, and as such, is not of much consequence. In addition, due to the large size of objects in the LOS, allocation requests can be aligned such that the overall space wastage is not significant compared to LOS utilization.

The authors observed that for segregation to have any benefit, large objects should be at least 1 KB in size. This implies that moving a small number of large objects to the LOS results in best performance. Moving large objects with internal reference fields to the LOS is also better than retaining them in the non-LOS portion of the heap. Although, there is per-byte cost associated with these objects during a non-copying LOS collection, the performance advantage due to segregation outweighs the overhead. Also, they concluded that the choice of non-copying collectors for the LOS does not seem to matter. Both mark-sweep and the treadmill algorithm perform almost equally.

#### 3.3.2 Pretenuing

Pretenuing [16, 26] attempts to allocate long-lived objects directly into heap regions that are collected less frequently (or not at all), in order to avoid copying overhead. We will discuss Blackburn et al's [16] pretenuing mechanism, since it purports to be applicable across applications and collection algorithms.

Blackburn et al presented two pretenuing mechanisms – *build-time* pretenuing, which attempts to combine pretenuing decisions across benchmarks so as to make it application dependent, and *application-specific* pretenuing, in which, profile advice that is specific to an application is used to guide the pretenuing deci-

sion for that application.

Blackburn et al used object age (normalized to maximum live data size) and time of death to guide pretenuring advice. They recorded all allocations, pointer mutations and objects death for a run of an application. Death information was obtained by frequently triggering full heap garbage collection cycles. Objects were classified into short-lived, long-lived, and immortal based on their time of death (a process called *binning*, so named because objects were classified into “bins”). Allocation sites were classified into short, long, and immortal, based on *homogeneity* of the type of objects that are allocated at a site. Allocation site classifications from different program executions were weighted and combined to form a combined classification. The build-time pretenuring decision is based on this classification of allocation sites. The formulation of the application-specific pretenuring decision is straightforward.

Blackburn et al measured application performance for build-time pretenuring, application-specific pretenuring, and combining both build-time and application-specific pretenuring. The authors also compared their results against Cheng et al’s [26] pretenuring, but they claimed that while Cheng et al’s work was application and collector specific, their work was somewhat application-independent. While measuring the effectiveness of build-time pretenuring for a particular application, the authors *do not* include information obtained by profiling that application, in the pretenuring advice. The authors showed that build-time pretenuring improved application performance in general, from very little for large heaps to an average of 8% for tight heaps. Application-specific pretenuring improved performance by 3.5% on average, for tight heaps. Combining the two mechanisms decreased garbage collection times by 20 to 32% for many heap sizes, and improved application performance by 7% on average, for tight heaps. The authors also argued that their application-specific pretenuring mechanism was on par with Cheng et al’s work across all heap sizes, and much better for tight heaps.

### 3.3.3 The Beltway Framework

Blackburn et al’s Beltway garbage collection framework [13], claims to include many important modern garbage collection concepts – the weak generational hypothesis and infrequent collection of old objects, older-first garbage collection [59], incrementality to improve responsiveness by decreasing pause time, and copying for compaction and improved locality.

The Beltway framework provides incrementality by using an *increment* as the unit of collection. Increments are arranged on *belts*, and collected sequentially, in a FIFO manner. A promotion policy is used to determine where to copy survivors.

The Beltway framework can be used to compose different configurations that correspond to well-known garbage collectors, including a non-generational semi-space collector [25], Appel’s generational copying collector [3], and an older-first collector [59]. More importantly, two new collectors are presented – the Beltway X.X, and the Beltway X.X.100. The Beltway X.X collector consists of two belts, each containing increments with a maximum increment size X.

A belt corresponds to a generation, and increments add incrementality to a generation, therefore the entire generation is never scavenged in one collection. Belts are assumed to be sequentially numbered. The lower belt corresponds to the nursery, and objects are allocated from this belt. When the lower belt is full, the oldest increment in the lower belt is scavenged and the survivors are copied to the youngest increment in the higher belt (which corresponds to the old object space). When the higher belt becomes

full, the oldest increment in the higher belt is scavenged, and survivors are promoted to the youngest increment in the *same* belt.

As such, Beltway X.X provides incrementality to Appel-style generational copying collection, with the parameter X indicating the level of incrementality. However, Beltway X.X does not guarantee that all garbage will be collected, even eventually. This is because it fails to collect cyclic garbage that spans more than one increment.

Beltway X.X.100 guarantees eventual completeness, i.e., a guarantee that all garbage will be collected, including large cyclic data structures, although not in one garbage collection cycle. It does so by adding a third (highest) belt, with a single increment. This increment can grow to be as large as the total size of the heap. Objects that survive collection in the two lower belts are promoted to the third belt. The third belt is collected only when it occupies the entire heap.

Blackburn et al also discussed implementation details for the framework, including use of frames, a write barrier implementation, and a dynamic copy reserve, which varies at runtime. The dynamic copy reserve is used to ensure that there is always enough space to copy live data. We will not discuss these details here. Instead, we will discuss some interesting results. First, the authors showed that the Beltway configuration that corresponds to a Appel-style generational collector, shows similar performance to the standard implementation of a Appel-style collector in the Jikes RVM. They also showed that an Appel-style generational collector improves performance by about 50% over a fixed sized generational copying collector. They then considered their new Beltway collectors and showed that Beltway X.100.100 is not too sensitive to increment size. Finally, the Beltway X.X.100 collector outperforms the Appel-style collector by an average of 5 to 10%, to a maximum of 35% for tight heaps. This is because Beltway X.X.100 is incremental and requires a smaller copy reserve area.

## 3.4 Garbage Collection Performance Evaluation

In this section, we will discuss some studies that compare the performance of different garbage collection techniques. In addition, we will also examine a study of the memory behavior of a set of well known Java benchmark applications.

### 3.4.1 Allocation Behavior of SpecJVM98 Benchmarks

Dieckmann et al [29] studied the allocation behavior of programs in the SpecJVM98 benchmark suite [57]. These benchmarks are used almost invariably by all researchers for measuring and comparing garbage collector performance in Java [13, 15, 53, 5, 11], and as such, a study of their allocation and memory behavior is very important and useful. Apart from total bytes allocated by each benchmark program, the authors studied heap size and object lifetimes, heap composition, reference density, and the effect of object alignment on heap space. The authors selected six out of the eight SpecJVM98 benchmarks – *compress*, a file compression/uncompression utility, *db*, a simple database management system, *jack*, a parser generator, *javac*, the JDK 1.0.2 Java compiler iterating over the source code of *jess*, *jess*, an expert system shell, and finally, *mtrt*, a multi-threaded ray-tracing application.

Their experimental setup consisted of a trace gathering utility that generated a trace file. This trace file served as input to a simulator, which simulated allocations, pointer assignments, and garbage collections. Various statistics were gathered by the simu-

lator during a run. Time was measured in terms of the amount of data allocated (in MB). The simulator performed a garbage collection cycle after every 50 KB of allocation.

Some observations regarding the benchmark programs are as follows. Most SpecJVM98 programs, except jack, allocate over 100MB of data, but they can all execute in a minimum heap size of less than 8MB. The maximum live heap size for all applications is established about one quarter of the way into their execution. After this maximum size is established, the amount of live data is kept fairly constant till the end of the run, either because the same objects are kept alive, or because objects are continuously discarded and allocated in phases.

The age distribution patterns for the benchmarks seem to confirm the weak generational hypothesis. However, the authors noted that this effect is not as pronounced, as it is in other languages like Standard ML of New Jersey and Smalltalk. They recommended that the young object space should be larger than for functional languages. In addition, the age distribution was found to be application-dependent.

Dieckmann et al measured reference density in order to determine the amount of work that needs to be performed during object scanning at garbage collection time. They found that applications allocate few reference fields, both during startup and at a later time. However, this behavior is also somewhat application-dependent. A heap composition study revealed that most data allocated by many applications consists of very few types, with one or two types dominating the rest. However, these types may not be same across applications. Also, basic types seem to have different age distributions. In addition, most long-lived data is of a single type.

The average object size across all applications was found to be quite small. It is beneficial to keep the size of the object header to a minimum, since, adding an extra word to the object header would increase the space overhead significantly. The authors also measured the extra space overhead due to alignment of objects on a 8-byte boundary. Since all applications allocate a large amount of small objects, the space overhead is considerable (about 19%).

### 3.4.2 Comparison of Garbage Collection Algorithms

Smith et al [55] compared the performance of the well-known conservative Boehm-Demers-Weiser (BDW) mark-sweep collector [18] and a mostly-copying collector [12]. As mentioned previously, programming languages for virtual machines are strongly typed, and consequently, garbage collectors are fully type accurate. As such, we will not consider the implementation of these collectors in detail, but even so, some conclusions drawn by this paper are quite interesting and relevant.

The BDW collector is a mature technology and is the most widely used conservative collector. The authors implemented a mostly-copying collector prototype and compared it against the BDW collector. The mostly-copying collector does not have a contiguous heap space divided into two semispaces, like most copying collectors. Instead it maintains a linked list of *pages*.

An object is not actually copied to the to space during a collection, but rather, the page containing the object is linked to a to space list. The major advantage of this organization is that fast allocation can be performed (faster than mark-sweep's free list allocation, although not as fast as a true copying collector's fast bump-pointer allocation). Objects are allocated from a single page in a contiguous manner until the page is full. This is similar to a traditional copying collector.

The authors showed that this prototype mostly-copying collector can be competitive with the BDW collector. The authors be-

lieved that a more sophisticated technique may lead to up to a 20% improvement in collection time over their prototype. More relevant is the authors speculation that improvements in the implementation, and presence of type information might make the mostly-copying collector competitive with copying collection, especially for applications with a high allocation rate.

Brecht et al [21] examined how to control garbage collection frequency and heap growth. Since they used the Boehm-Demers-Weiser conservative collector for this study, we will not discuss their implementation and methodology. Instead, we will summarize some of these important findings. The authors attempted to correlate heap growth with the amount of memory available to the operating system. After conducting many experiments with different garbage collection frequencies, and using different *free space divisor* values (this parameter controls heap growth), the authors concluded that frequent garbage collections are detrimental to application performance. If sufficient memory is free, garbage collections should not be performed and the heap should be grown aggressively. The heap can be grown by requesting more memory from the operating system. The authors also found that when the amount of memory becomes low, some memory should be made available, so that paging can be avoided. For this purpose, garbage should be collected aggressively (more frequently), and the heap should not be grown much. When the amount of memory is low, the *initiation* of garbage collection becomes expensive. Consequently, garbage collection should not be performed if the amount of memory reclaimed in successive collections is very small.

Zorn [66] compared the CPU overhead and memory costs of a generational mark-sweep collector and a generational copying collector. He extended pure copying and pure mark-sweep collectors to use generations.

The generational copying collector maintains a copy count per object, indicating the number of times the object has been copied. When the copy count reaches three, the object is promoted. Allocation is via a simple and fast pointer increment. The generational mark-sweep collector is a non-compacting algorithm, with object marking performed by setting a mark bit in a bitmap (instead of storing it in the object header). This improves locality of the mark-test-clear operation and enables efficient sweeping. Sweeping is not performed directly after the marking phase is complete. Instead, it is deferred and performed during object allocation in an incremental manner. Promotion in the generational mark-sweep algorithm is done *en-masse*, i.e. all objects from the young object space are promoted after three minor collections.

Zorn evaluated the two collectors using a trace-driven simulator called MARS. He found that the allocator used with the generational copying collector is faster than the one used with the generational mark-sweep collector. As such, mark-sweep has a higher garbage collection independent cost. However, copying collection has a higher garbage collection dependent cost resulting from the overhead due to copying. In addition, mark-sweep requires much less memory (20% less, on average) to achieve the same page fault rate as the copying collector. This is due to the fact that the copying collector requires a large copy reserve area, while the mark-sweep collector collects in place. Zorn also noted that the mark-sweep algorithm's *en-masse* promotion policy has a significant space overhead when promotions are frequent. Although Zorn's results seem quite obvious today, his study is quite important since it is one of the first that shows that mark-sweep collection is not always inferior to copying collection, contradicting what was believed popularly.

Most garbage collected runtime environments use a single garb-

age collector across all applications. That a single collector may not be well suited for all applications is a well known fact.

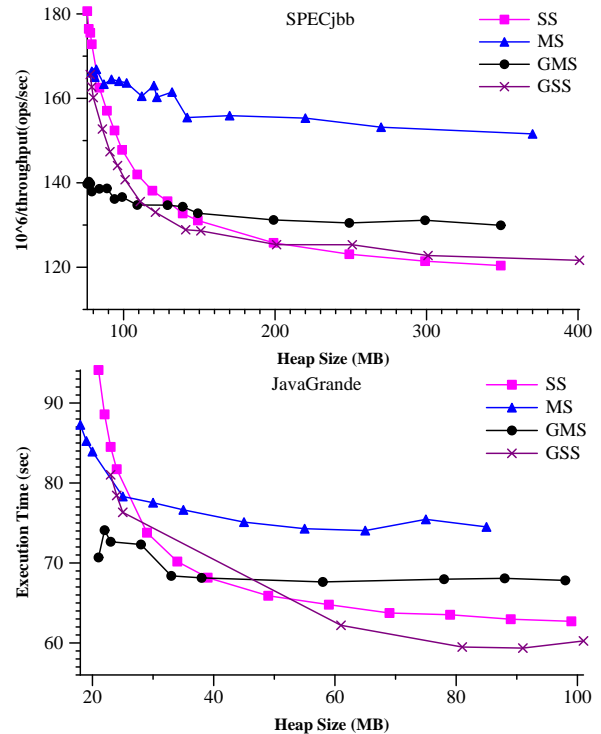
Attanasio et al [5] compared the performance of several different garbage collectors that they have implemented in the Jikes RVM (which was then known as the Jalapeño virtual machine). They compared a semispace non-generational copying collector, a mark-sweep non-generational collector, generational versions of these two basic types of collectors, and a hybrid collector that makes use of mark-sweep for the mature space and copying collection for the nursery. The collectors execute as multiple garbage collection threads, running in parallel during collection time, i.e., they are parallel collectors. They are however, not concurrent and the mutator must be paused during collection. A load balancing queue balances scanning work between the collectors. These collectors were targeted towards SMP machines running server applications.

Attanasio et al measured collector performance across six benchmark programs. They found that the hybrid collector has the smallest heap residency, due to minor collection of young objects that die quickly, and the use of a more space efficient (compared to copying) mark-sweep collector for the old object space. However, *no one* collector enables the best application performance. In addition, *every* collector performs the best for at least one application. The mark-sweep collectors outperform the copying collectors for applications with a large working set of data, and consequently, more copying overhead. The hybrid collector does not always perform better than the rest, mainly because the write barrier overhead offsets the performance improvement resulting from less object scanning, for some applications. Fragmentation of the old object space can also play a role in deciding application performance, as is demonstrated by the performance of some applications that operate best with the generational copying collector (and not generational mark-sweep or the hybrid).

Attanasio et al also found that mark-sweep performs fewer collections than copying collectors. Minor collection is more expensive for the hybrid collector, as compared to the generational copying collector, since allocation during promotion uses a free-list allocator in the former case, and a fast bump-pointer allocator in the latter.

The application-specific nature of garbage collection was further corroborated by Fitzgerald and Tarditi [32]. They measured performance for a set of 20 Java benchmark programs, across several different garbage collectors. The benchmark programs were compiled with the Marmot optimizing Java to native compiler [31]. The garbage collectors chosen were a null collector, which allocates but never collects, a non-generational copying collector that uses Cheney’s traversal algorithm [25], and a two-generation generational copying collector with multiple variants. The variations are due to different write barrier implementations – a filtered write barrier with a sequential store buffer, and two different card table implementations, and an unfiltered write barrier with a card table.

Fitzgerald et al observed that *every* collector is the worst choice for at least one application. At least two benchmarks perform more than 15% worse than they would have with a different (more appropriately chosen) collector. Varying the collector type has no visible performance effect for only two benchmarks out of 20. Contrary to popular belief, generational collection does not always outperform non-generational collection. For some benchmarks, generational collection reduces collection time enough to recover the time lost due to write barrier overhead. However, for other benchmarks, it performs worse than the non-generational algorithm by 15 to 20%. The authors found that whether or not to



**Figure 2: Application performance for different GC systems and heap sizes (lower is better).**

use generational collection is the most significant choice affecting performance. The authors also showed that runtime systems can use different pre-compiled binaries (that implement the virtual machine) for different applications. They stated that it is difficult or impossible for a system to replace its garbage collector at runtime. In the remainder of this paper, we will show that it is indeed possible to replace garbage collection algorithms at runtime.

#### 4. DYNAMIC SELECTION OF APPLICATION-SPECIFIC GC

The next-generation of high-performance server systems must enable continuous availability and high-performance to gain widespread use and acceptance. Due to the portability, flexibility, and security features enabled by the Java programming language and its execution environments, a number of high-end server systems now employ Java as the implementation language for application and execution servers [40, 33, 52]. These systems run a single virtual machine (VM) image continuously so that applications and code components can be uploaded and executed as needed by customers (for customization, collaboration, distributed execution, etc.).

Given this model (single VM and continuous execution) and existing JVM technology, a single, general-purpose collector and allocation policy must be used for all applications. However, many researchers have shown that there is no single combination of a collector and an allocator that enables the best performance for all applications on all hardware and given all resource constraints [5, 32, 66]. Figure 2 confirms these findings. The graphs show performance (lower is better) over heap size for the SPECjbb [58] and the JavaGrande [42] benchmarks, executing within the JikesRVM [2].

Figure 2(a) shows that for SPECjbb, the semispace (SS) collector, performs best for all heap sizes larger than 200MB, the generational/semispace hybrid (GSS) performs best for heap sizes 120-200MB, and the generational/mark-sweep hybrid (GMS) performs best for heaps smaller than 120MB. In Figure 2(b), GMS is the best for small sized heaps, SS performs well for heap sizes 40-53MB, and GSS performs best for heap sizes larger than 53MB. We refer to the point at which the best-performing GC system changes as the *switch point*.

To exploit this execution behavior that is application-specific and dependent upon the underlying resource availability, we extended the JikesRVM adaptive optimization system for Java, to enable dynamic switching between GC systems. The goal of our work is to improve performance of applications for which there exists GC switch points, without imposing significant overhead.

#### 4.1 The JikesRVM and the JMTk

The JikesRVM [2] is an open-source, dynamic and adaptive optimization system for Java that was designed and continues to evolve with the goal of enabling high-performance in server systems. The JikesRVM (version 2.2.0+) implements the Java Memory Management Toolkit (JMTk) that enables garbage collection and allocation algorithms to be written and “plugged” into the JikesRVM. The framework offers a high-level, uniform interface to the JikesRVM that is used by all algorithm implementations. We refer to the combination of an allocation policy and a collection technique as a *GC system*. This corresponds to a *Plan* in the JMTk terminology. The JMTk allows users to implement their own GC systems easily within the JikesRVM and to perform an empirical comparison with other existing collectors and allocators. When a user builds a configuration of the JikesRVM, she is able to select a particular GC system for incorporation into the JikesRVM image.

Each GC system in the JikesRVM is implemented via a *Plan* and *Policy* class. Each GC system is linked to a virtual memory resource (*VM\_Resource*) which binds the allocation region to particular virtual address ranges. In addition, the system monitors (polls) the remaining free memory space and initiates collection as needed. Collection proceeds according to the associated policy. A policy consists of a set of classes that implement the type of collector (mark-sweep, semispace, generational, etc.) and the allocator (free-list, bump-pointer, etc.).

The four GC systems that we consider in this work are Semispace (SS), Mark-sweep (MS), a Generational Semispace Hybrid (GSS), and a Generational Mark-sweep Hybrid (GMS). These systems use stop-the-world collection and hence, require that all mutators pause when garbage collection is in progress.

The SS system consists of a virtual memory resource that maps the heap address range to a contiguous block of memory, and a *bump-pointer allocator* that allocates memory in contiguous chunks from the virtual memory resource. The virtual memory space is divided into two half-spaces, equal in size: the *from* and *to* semispaces. Memory is allocated from only one semispace at any time, and hence, the usable virtual address space is half of the total space. During a collection, live objects are copied from the *from-space* to the *to-space*. At the end of the collection, the roles of the semispaces are reversed. The SS system also includes a separate space for allocation of large objects. Large objects are allocated by a sequential first-fit free list allocator and collected using the mark-sweep technique.

In the MS system, memory is allocated from the mark-sweep space using free-list allocation, like that for large object allocation. Collection is a two-phase process that consists of a mark

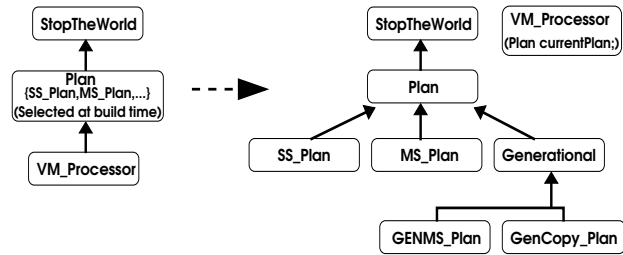


Figure 3: Original and New (Dynamic Switch-Enabled) JikesRVM JMTk Class Hierarchy

phase in which live objects are marked, and a sweep phase in which unmarked space is reclaimed.

The GSS system makes use of well-known generational GC techniques [3, 62]. Young objects are allocated in a variable-sized nursery space using bump-pointer allocation. Upon a minor collection, the nursery is collected and the survivors are copied to the mature space. The mature space is collected by performing a semispace copying collection, following a minor collection, as needed. This process is referred to as a major collection.

The GMS system also employs a generational model: There is a nursery which holds young objects and a mature space for the old objects. However, the mature space is collected using a mark-sweep algorithm and allocated using free-list allocation.

The MS and GMS systems do not use a large object space by default. All GC systems include an immortal space that holds the JikesRVM system classes. Immortal space is allocated using the bump-pointer technique and this space is never collected.

We extended the JikesRVM to switch between SS, MS, GMS, and GSS at runtime. In the following section, we describe our switching framework.

#### 4.2 Dynamic GC Switching System

The JikesRVM *Plan* class implements the allocation and collection strategies; the source-code implementation for this class is stored in a sub-directory that corresponds to each individual GC system that is supported by the JikesRVM. For example, if a user chooses to use the semispace collection system, she builds the appropriate JikesRVM configuration that indicates this. The build process copies the *Plan* class from the semispace sub-directory so that it is used as the implementation for the *Plan* in the system. By default, only one *Plan* class can exist in a JikesRVM image. The only way to change *Plans* (to use a different garbage collector) is to build another image using a different JikesRVM configuration.

Our extension to the JikesRVM requires that multiple GC systems be included in a single system image. To enable this, we implemented a generic *Plan* class, from which all specific GC system classes derive, e.g., *SSPlan*, *MSPlan*, *GMSPlan*, and *GSSPlan*. Each of these plans are instantiated in a single image of our system. Figure 3 shows the JikesRVM JMTk class hierarchy before and after our extensions.

We inserted a global field called *currentPlan* into the class that implements the GC system interface to the JikesRVM (*VM\_Interface*). This field identifies the GC system that is currently in use. At all times, an object instance of each GC system (a plan instance) is available in our system. The current default allocator that is used, depends on the current collection system. The *Plan* class invokes a static allocation routine according to the GC system indicated by *currentPlan*. When a switch occurs, the instance of the appropriate collector becomes the global instance of the

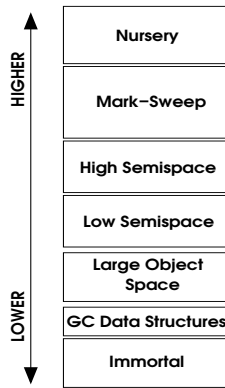


Figure 4: Address Space Layout in the Switching System

system.

The immortal space and large object space within our system are shared across all GC systems. Since the extant versions of MS and GMS do not implement a large object space by default, we extended both to do so. Objects larger than 16KB are allocated from the large object space. To support multiple GC systems, we require address ranges for all possible virtual memory resources to be reserved. We try to make efficient use of the virtual address space and to overlap as many address ranges as possible (Figure 4). Note that these address ranges are mapped to physical memory lazily (as it is needed), in 1 Megabyte chunks.

Switching between GC systems requires that all mutators be suspended to preserve consistency of the virtual address space. Since the JikesRVM collectors are all stop-the-world, the system implements the necessary functionality to pause and resume mutator threads. We extended this mechanism to implement switching.

Our implementation, however, does not require garbage collection to be performed for all switches. For example, a switch from MS to GMS or SS to GSS only requires that future allocations come from the nursery area. As such, we only need to perform general bookkeeping to record the current plan. Similarly, when we switch from a generational to a non-generational collector, we need only perform a minor collection in most cases. After the switch, we suspend collector threads and resume the mutators, as is done during the post-processing of a normal collection.

Although the switching process is specific to the old and the new GC systems (as we shall describe below), we provide an extensible framework that facilitates easy implementation of switching from any GC system to any other, existing or future, that is supported by the JikesRVM JMTk.

**Mark-sweep (MS) to Generational Mark-sweep (GMS).** In our implementation, MS and GMS share the same free-list resource and virtual address space (the mark-sweep space for MS and the mature space for GMS). The switch from the MS GC system to a GMS system does not require a collection. We need only to update the *currentPlan* field to point to the GMS system. Thus, there is no additional cost involved with the switch other than stopping all mutators, updating a field, and resuming the mutator threads.

**Generational Mark-sweep to Mark-sweep.** To switch from the GMS GC system to the MS system, we perform a minor collection so that all live objects from the nursery are copied to the mature space. We then set the *currentPlan* field to point to the MS system. Thus, at the end of the switch, the nursery is empty and the mature

space is now the MS system’s mark-sweep space.

**Semispace (SS) to Mark-sweep or Generational Mark-sweep.** To switch from the SS to the MS GC system or to the GMS system, we perform a semispace collection. However, instead of copying survivors to the empty semispace, we copy them to the mark-sweep space.

**Mark-sweep or Generational Mark-sweep to Semispace.** For the MS to SS switch, as we mark live objects in the mark-sweep space, we forward them to the semispace resource. However, this switch is more complex than the ones previously described. The use of object forwarding during a Mark-sweep collection requires us to maintain multiple states per object. We detail this process and its implementation in Section 4.3.

Switching from a GMS to a SS GC system is similar to the MS to SS switch. We perform a major collection and copy survivors from the nursery as well as live objects from the mature space to the semispace.

**Semispace to Generational Semispace (GSS).** In our implementation, the two half-spaces of the SS GC system are shared with the GSS GC system. The cost of switching from the SS GC system to GSS is similar to the cost of switching from the MS GC system to the GMS system. No garbage collection is required to effect the switch.

**Mark-sweep/Generational Mark-sweep to Generational Copying.** To change from the MS or the GMS GC system to the GSS system, we need to perform steps similar to the MS/GMS to SS switch. In fact, we share the same code, and hence we were able to implement this switch with minimum additional programming overhead.

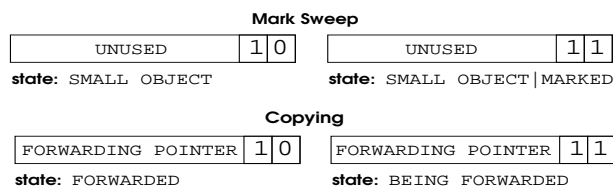
**Generational Copying to Mark-sweep/Generational Mark-sweep.** This switch is similar to that of SS to MS or GMS. However, we need to copy over objects from the nursery to the shared free-list region, in addition to copying objects from the GSS mature space to the shared region.

Unlike previous work, we are thus able to dynamically switch between GC systems that use very different allocation and collection strategies. Although, for the results presented in this paper, we switch GC systems only once during a run of an application, our framework is completely general. Coupled with dynamic decision mechanisms, we can switch from any GC system to any other, multiple times. We next describe details that are specific to our implementation within the JikesRVM.

### 4.3 Implementation Details

We employed four primary implementation strategies to make our system compatible with the existing JikesRVM infrastructure and to make it as efficient as possible. They are, maintaining multiple states for forwarded objects in a mark-sweep collected space, unmapping unused memory, inlining of allocation routines, and using a single, shared write barrier implementation.

As mentioned in the previous section, to switch from a GC system that uses a mark-sweep space to a GC system that uses a contiguous semispace, we need to maintain state for the mark-sweep process as well as for the process of forwarding objects to the semispace. In the JikesRVM, the mark-sweep collector requires two bits in the object header: the *mark bit* to mark live objects and the *small object bit* to indicate that the object is a small object. The use of the *small object bit* is specific to the free-list implementation in the JikesRVM. Since memory allocation requests are aligned on a 4-byte boundary in the JikesRVM, the lowest two



**Figure 5: Examples of Bit Positions in the Object’s Status Word**

bits in an object’s address are always 0. Hence, the *mark bit* and the *small object bit* can be encoded as the lowest two bits in the object’s status word, which is stored in the object header.

The copying process uses two additional states. An object is marked as *being forwarded* while it is being copied. After it is copied, the object is marked as *forwarded* and a forwarding pointer to the object’s new location is set in the old object’s header. The *being forwarded* state is necessary to ensure synchronization between multiple collector threads. These two states are stored in the status word of each object, along with the forwarding pointer. Thus, the lowest two bits in the object’s status word have different purposes depending on the collector that the JikesRVM is configured with, while building the boot image. For example (Figure 5), if the JikesRVM is built with the mark-sweep GC system, the value `0x2` indicates that the object is a small object. However, if instead, the semispace collector is used, this state indicates that the object has been forwarded to the to-space during a collection. Similarly, if the lowest two bits are set, it signifies that the object is a small object and has been marked live by the mark-sweep collector; the same state indicates to a semispace collector thread that the object is currently being forwarded by another thread.

Since we forward marked objects to the semispace, we need to support all four distinct states, along with the space required for the forwarding pointer. To account for the two additional bits required, we use a technique known as bit-stealing [8]. The object header stores a pointer to its Type Information Block (TIB). In the JikesRVM, TIBs store information about the object’s class (including the virtual method table). A TIB is defined to be aligned on a 4-byte boundary. Hence, we can use the two lower-most bits of the word that points to the TIB, to mark the *being forwarded* and *forwarded* states during the copying process.

The second feature we implemented is memory unmapping. The reference JikesRVM implementation uses on-demand memory mapping of the virtual address space. To use physical memory efficiently, we unmap the memory mapped space that we won’t use when we switch to a new collection system.

A third mechanism that we use to improve the efficiency of our system is the use of static methods to avoid dynamic dispatch where possible. Allocation in our switching system occurs via a static routine, *Plan.alloc* in the *Plan* class. This routine performs a check on an integer value representing the allocator to use (a *switch...case*). We maintain a global integer field called *CURRENT\_DEFAULT\_ALLOCATOR*, which indicates the current default allocator to use. *Plan.alloc* invokes the appropriate allocation routine based on the value of *CURRENT\_DEFAULT\_ALLOCATOR*. *Plan.alloc(...)* can be inlined into the program to reduce the overhead of function calls for allocation. The individual allocation methods are not inlined into *Plan.alloc* since the particular method can change dynamically. The reference JikesRVM implementation contains only a single plan and as such, all allocation routines can be inlined.

A second source of overhead, other than our inability to inline

Program	Annotated GC Selector
compress	if (heapsize $\geq$ 50MB) GSS else GMS
db	if (heapsize $\geq$ 30MB) SS else MS
jack	GMS for all heap sizes
javac	GSS for all heap sizes
jess	GMS for all heap sizes
mpegaudio	SS for all heap sizes
mtrt	if (heapsize $\geq$ 40MB) GSS else GMS
JavaGrande	if (heapsize $\geq$ 72MB) GSS else GMS
OptComp	if (heapsize $\geq$ 118MB) SS else GMS
SPECjbb	if (heapsize $\geq$ 150MB) SS else GMS

**Table 1: Annotated garbage collection selection decisions for each benchmark**

each of the individual allocation routines, is a universal write barrier, which is necessarily introduced by our system. Write barriers are used to record pointers for generational collectors [14, 37] since heap areas are independently collected. Cross-generation (old-to-young) pointers must be tracked so that they can be traced during collection of the young (nursery) heap space. Since our system can switch at any time to a generational garbage collector, we must always insert write barriers. However, to make this process as efficient as possible, we use a single, shared write barrier for all GC systems. In our system, the nursery always occupies the highest virtual address range. Hence, we require only a single check to determine if the young object reference is in the nursery. We evaluate the impact of these overheads in Section 6 and discuss solutions for reducing each.

## 5. GARBAGE COLLECTOR SELECTION

By implementing functionality to switch between collection systems while the JikesRVM is executing, we can now select the “best-performing” collection system for each application that executes using our system. To show the efficacy of this functionality, we consider annotation-guided GC system selection and automatic selection based on simple heuristics.

### Annotation-Guided Selection

To enable annotation-guided GC selection, we analyzed application performance off-line using the different JikesRVM GC systems. We considered a number of different heap sizes and program inputs (SpecJVM98: input 10 and 100, SPECjbb: one and two warehouses, JavaGrande: section3/AllSizeA and AllSizeB). We extracted, for each heap size, the best performing GC system across inputs. At each point, if the GCs selected were different across inputs, we identified the GC that imposed the smallest percent degradation. We then identified a cross-over threshold for each program, i.e., the heap size below which the most-commonly selected GC changes. In some cases, the GC never changes, i.e., there is no switch-point. The annotated values are shown in Table 1.

We annotated programs with GC identifiers for a range of heap sizes. We used a 4-byte annotation in each class file of an application containing a main method. We inserted annotations into class files using an annotation language and a highly compact encoding that we developed in prior work [44]. Upon initiation of dynamic loading of the first application class file, JikesRVM switches to the collection system specified. If an annotated GC system is not available, the default JikesRVM collection system is used (currently, GMS). Since the best-performing collection system may depend on the underlying architecture (memory size, cache lev-

els, cache sizes, register count), we can also incorporate different architectures as part of our profile collection and annotation. For this work, we focus solely on the x86 architecture.

### Automatic Switching

To automate the switching process, we monitor execution behavior while the program is running to determine when to switch, by using simple heuristics based on maximum heap size and heap residency following GC. The empirical evaluation of our annotation-guided system indicated that, across inputs, the best performing collector is consistently GMS for small heaps and GSS for large heaps. If the heap availability should change, e.g., to make room for concurrent execution of other programs, our system can automatically switch to GMS or GSS accordingly. Automatic switching avoids the need for both off-line profiling and program annotation.

In addition to determining what GC system to switch to, we must also identify *when* to switch. Some possible options include heap residency thresholds, GC frequency thresholds, and allocation behavior. As an experiment, we implemented the above heuristic (GMS/GSS switching with a 90MB heap size threshold) using a heap residency threshold of 60%. As such, given any application, our system waits until the live data following a collection exceeds 60% of the available heap size. At which point, the system checks whether the maximum heap size is greater than 90MB, and if so, switches to GSS. The system uses GMS as the initial, default GC system. As such, no switching is needed if the maximum heap size threshold is not exceeded. Use of a residency threshold enables us to use two different collectors for different phases of program lifetime: program startup and steady-state.

We acknowledge that this is a very simple implementation of automatic GC selection. We include it as an example of how our switching framework can be employed. We intend to study extensively techniques for adaptive switching as part of future work.

## 6. EVALUATION

To empirically evaluate the efficacy of our framework, we performed a series of experiments using a number of benchmark programs. We gathered results on a dedicated 2.4GHz x86-based single-processor Xeon machine (with hyperthreading enabled) running Debian Linux v2.4.18. We implemented our switching framework within the JikesRVM version 2.2.0 with jlibraries R-2002-11-21-19-57-19. We report performance results using the JikesRVM Fast configuration in which all methods executed are optimized upon initial invocation. We repeatedly executed the benchmarks through a test harness and report the average of the last 5 of 10 runs; therefore, compilation is not included in our data (except for the OptCompiler benchmark which exercises the JikesRVM compilation system). The benchmarks we consider are the SpecJVM98 suite, SPECjbb2000, the JavaGrande suite, and the JikesRVM optimizing compiler (first harness run of JikesRVM executing the SpecJVM98 javac benchmark).

### 6.1 Results

We first present results that show the efficacy of switching using annotation-guided GC selection. The results are shown in Figures 6 and 7 for a number of benchmarks. The x-axis is heap size in megabytes and the y-axis is total time (in milliseconds) for program execution. For SPECjbb, the y-axis is the inverse of the throughput reported by the benchmark; we report this metric to maintain visual consistency with the execution time data, i.e., lower numbers are better. As described in Section 5, we selected the best performing GC system for a range of heap sizes

Overhead due to Universal Write Barriers And Not Inlining Allocation Routines		
Benchmark	Overhead	
	Alloc Not Inlined	Universal Write Barrier
compress	-2.76%	-0.23%
jess	3.47%	4.50%
db	2.82%	2.24%
javac	-3.44%	-2.31%
mpegaudio	-6.45%	-4.14%
mrt	7.59%	0.35%
jack	6.65%	0.33%
OptComp	*-51.59%	*14.02%
JavaGrande	0.14%	-0.43%
SPECjbb	0.83%	0.27%
Geo. Mean	-6.25%	1.36%

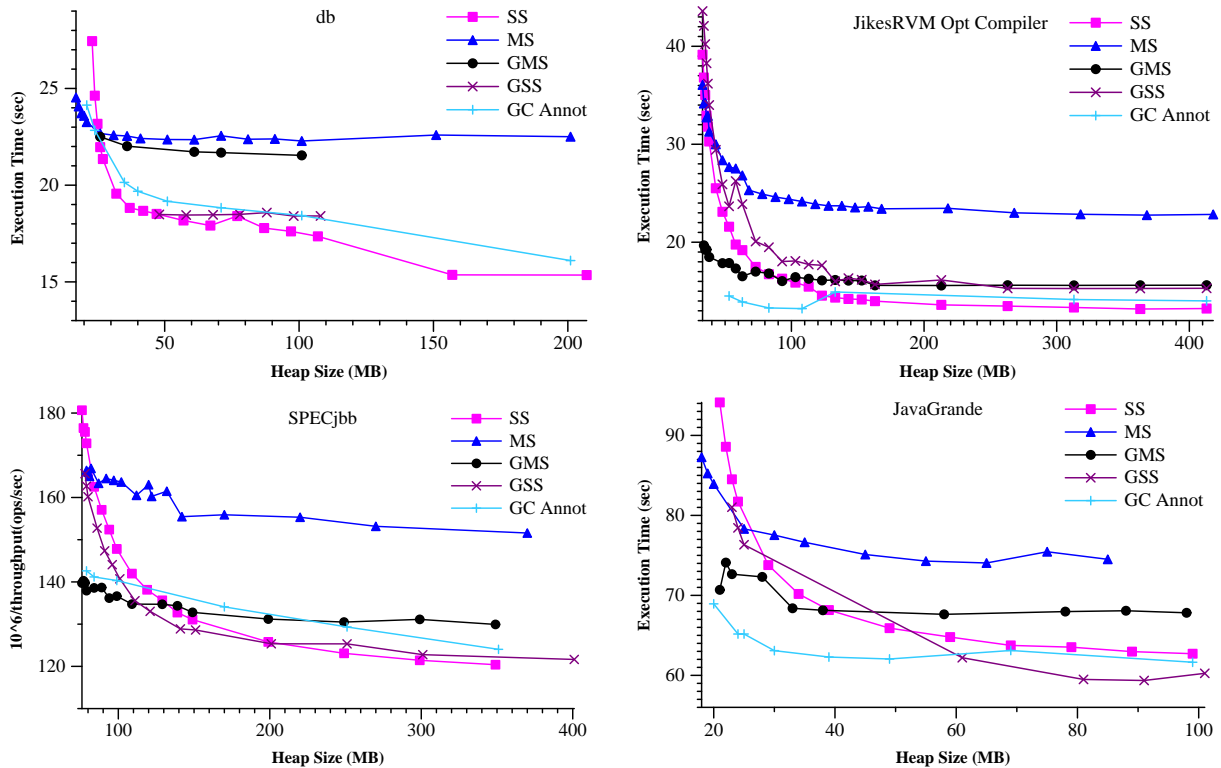
**Table 2: Overhead due to NOT inlining allocation routines (column two) and always adding write barriers (column three) for the SpecJVM98 benchmarks. Values represent mean percentage difference in performance over all measured heap sizes. The star (\*) indicates that the results of the three different switching system configurations are not directly comparable, since the amount of code compiled differs: the switching system does not inline allocation sites and it inlines write barrier code. In all other cases, compilation overhead is not included.**

by considering multiple inputs offline: SpecJVM98 – input 10 and 100, SPECjbb – one and two warehouses, JavaGrande – section3/AllSizeA and section3/AllSizeB). We identified the best performing GC system all inputs (see Table 1), for each heap size. For brevity, we present results only for input 100 for SpecJVM98, a single warehouse for SPECjbb, and input section3/AllSizeA for JavaGrande.

Each graph shows the performance of each GC system that we studied: MS, SS, GSS, and GMS. In addition, each shows the performance of our annotation-guided GC switching system. The results indicate that our framework is able to achieve performance that is similar to the best performing collector by making use of the annotated information. For cases in which there is no cross-over between optimal collectors, e.g., jess, mpegaudio, javac, our system maintains performance similar to that of the reference system.

The overhead introduced by our system is low for most benchmarks. The main source of overhead results from the loss of inlining opportunities for allocation sites. Since our system must dynamically check the type of GC system in use prior to deciding which allocation routine to invoke, we are unable to freely inline such sites. Another source of overhead, is the presence of a universal write barrier, which is necessary, since our system provides the ability to switch to a generational collection system, at any time.

Table 2 shows the mean percentage overhead due to not inlining allocation routines and always adding write barriers for the SpecJVM98 benchmarks, across all measured heap sizes (starting from the minimum to up to about 600MB). We ran these benchmarks using our framework, *without* ever switching; we used the SS collection system for all heap sizes (minimum to 200MB). We then calculated the mean difference between the execution times so obtained and the execution times using the same system (with switching implemented, but without actually switching) with allocation routines inlined (column 2) and without inlining the universal write-barrier (column 3).



**Figure 6: Performance Results.** For each benchmark, data for the reference system is shown (SS, MS, GMS, GSS). In addition, the data demarked with (+) shows the efficacy of annotation-guided GC system selection.

The star (\*) indicates that the performance of the switching system configurations are not directly comparable since the amount of code compiled differs: the switching system in column 2 does not inline allocation sites and it inlines write barrier code in column 3. This is only the case for OptComp for which we are analyzing compilation overhead; for all other cases, compilation overhead is not included. The mean values without the OptComp benchmark were 0.89% for column 2 and 0.04% for column 3.

The data indicates that our system does not impose significant overhead. We show execution times for a range of heap sizes for a representative set of benchmarks, in Figure 8 – to point out some anomalies. Again, this data shows the performance of the switching system when no switching is performed; it shows the impact of not inlining allocation sites and having to inline write barriers – for a non-generational semi-space collector.

The jess benchmark is representative of most others in the table (jess, db, mtrt, jack, JavaGrande, SPECjbb). These benchmarks actually show a performance degradation with inlining of allocation routines turned on for small heaps. This is due to the fact that in tight heaps, there are many garbage collection cycles. Inlining causes the code size to increase, which is detrimental to performance in small heaps, since a copying collector like SS, copies large code arrays between heap partitions. For large heap sizes, there are no garbage collections, and the performance improvement due to inlining, dominates.

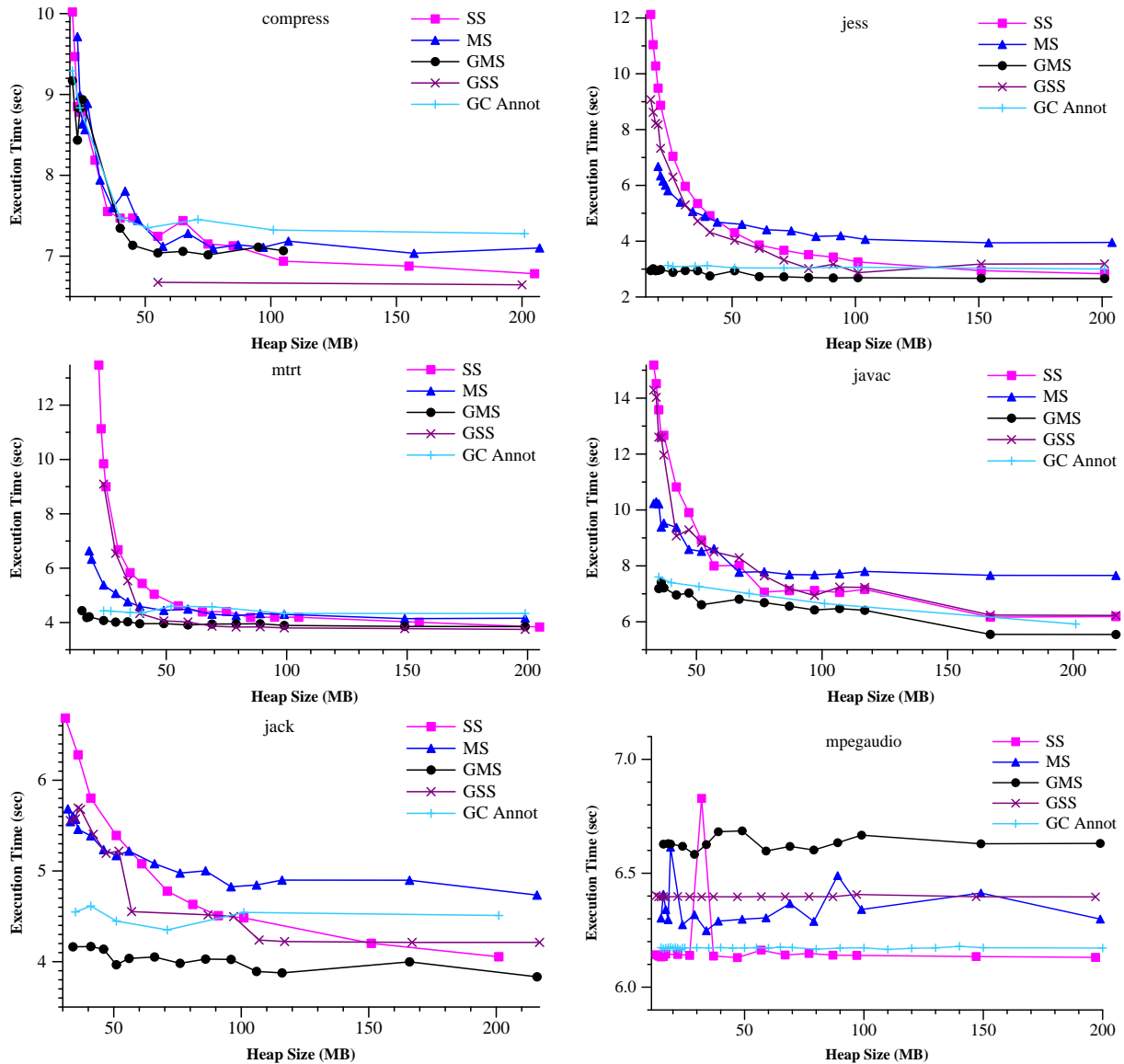
Jack exhibits the worst overall overhead, due to missed inlining opportunities for allocation sites. Inlining of allocation routines seems to always help performance for jack. Javac is also impacted by turning on inlining, however, in a way that is more similar to jess than jack. Inlining of allocation routines with small heaps hurts performance, since the average overhead due to dis-

abled allocation inlining is negative (in Table 2). These results indicate that we should also consider inlining as a parameter to annotation-guided and automatic GC switching.

The mpegaudio performance results are very anomalous and non-intuitive. When we inline allocation sites, performance degrades – and no compilation is being performed for these runs. We believe that the degradation is caused by instruction cache misses due to differences in code size. Mpegaudio allocates very little data and no GCs occur even for the minimal heap size. We have verified that disabling inlining does degrade performance for mpegaudio in the clean (non-switching) JikesRVM v2.2.0 using the semispace collector, as well as in more recent releases. This suggests that the anomalous behavior is due to some reason that is inherent in the nature of these benchmarks, and not a product of our framework.

For mpegaudio (and to a lesser extent, javac), removing write barriers *degrades* performance. We do have a good explanation for and are currently investigating this anomaly.

Not inlining allocation routines improves performance for small heaps since less code remains resident for manipulation by the collector. Our inability to inline can have another positive effect on program performance. For some programs, e.g., the JikesRVM Optimizing compiler benchmark, our switching system enables better performance than all reference collectors. This is due to reduced compilation overhead. The optimizing compiler aggressively inlines methods, including those for allocation. As a result, a large amount of time is spent repeatedly optimizing inlined code. This can be seen in Table 3 which shows the time in milliseconds for compilation in the reference system and our switching system. The overhead of our system is significantly lower for such cases because allocation sites cannot be inlined.

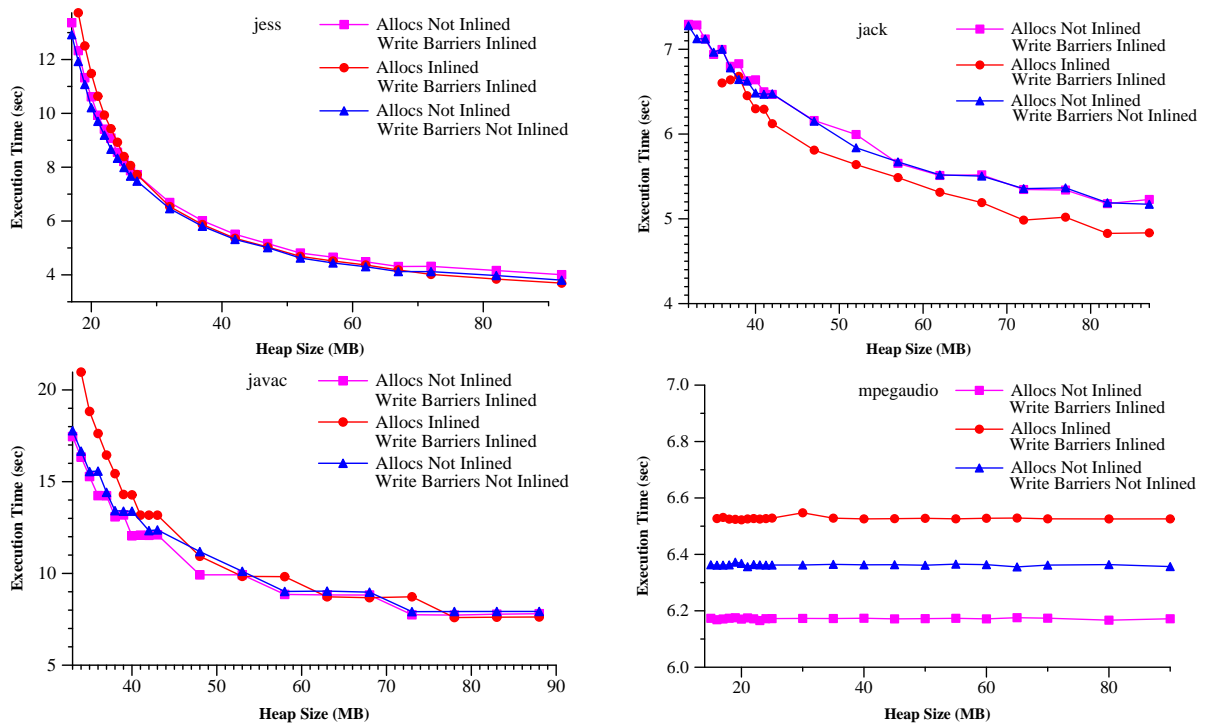


**Figure 7: Performance Results, continued.** For each benchmark, data for the reference system is shown (SS, MS, GMS, GSS). In addition, the data demarked with (+) shows the efficacy of annotation-guided GC system selection.

To reduce the overhead of our framework, we can specialize methods according to the currently selected GC system. That is, we can assume that the GC will never change, and as such, inline GC-specific allocation routines. Similarly, we can avoid inlining write-barriers if the current GC does not require them. However, when a switch occurs, we must invalidate the specializations. This requires simple recompilation for most methods. However, methods that are on the runtime stack will require on-stack-replacement (OSR). OSR enables replacing the stack frame of an invalidated method with another (unspecialized version). As part of future work, we plan to build upon existing work to enable both invalidation [36, 4] and on-stack-replacement [30, 36]. The trade-off of performing specialization is recompilation overhead and unoptimized execution time. We plan to empirically evaluate these trade-offs as part of future work.

To summarize our empirical evaluation of annotation-guided GC switching, we next show how our system reduces perfor-

mance hits taken when the “wrong”, i.e., worst-performing collector, is chosen. In addition, we show the average performance degradation over optimal selection. The data in Table 4 shows the average difference between our GC switching system and the best performing system at each heap size (column 2) and between our system and the worst performing system at each heap size (column 3). In parentheses, we show the average absolute difference in milliseconds; for SPECjbb the value in parenthesis is the difference in inverse throughput. Note that the data in this table does not compare our system against a single JikesRVM GC system; instead, we are comparing our system against the best and worst-performing GC system at every heap size. For example, for large heap sizes for the SPECjbb benchmark, the SS system performs best. For small heap sizes, GMS performs best. In this case, to compute percent degradation, we take difference between execution times enabled by our system and the SS system for large heap sizes, and our system and the GMS system for small heap sizes.



**Figure 8: Execution times for some benchmarks with our GC switching system always running the SS collector (squares), allocation routines inlined in the GC switching system (circles), and write barriers not inlined in the GC switching system (triangles), across different heap sizes**

Benchmark	Compilation Times (sec)			
	input1		input2	
	Reference	Switch	Reference	Switch
compress	1655.0	1302.0	1655.5	1306.5
jess	3572.0	2781.0	3595.5	2822.5
db	1906.0	1508.0	1891.0	1483.0
javac	6210.0	5671.0	6313.5	5877.0
mpegaudio	2349.5	2067.5	2346.5	2052.5
mtrt	2252.5	2008.0	2256.5	2011.0
jack	6347.0	4211.0	6343	4211.5
JavaGrande	2758.0	2631.5	2682.5	2628.0
SPECjbb	13014.0	8566.5	12978.0	8594.5

**Table 3: Compilation overhead introduced by JikesRVM dynamic compilation and optimization.**

In column two (Degradation over Best), some values are negative. In these cases, the performance of our switching system is better than that of all JikesRVM reference configurations across heap sizes, on average, i.e., our system improves performance over the best-performing GC system. This is due to the combined effect of missed inlining of allocation sites and our use of static method invocations that avoid the dynamic dispatch used in the reference system. Our annotation-guided system degrades performance over the best-performing GC system by 4% on average across benchmarks. In addition, it improves performance by 19% over the worst-performing GC system.

### Automatic Switching

In addition to annotation-guided GC selection, we also implemented automatic switching using a simple heuristic (based on 60% heap residency). Table 5 shows the average performance

Benchmark	Average Difference Between Best & Worst GC Systems	
	GCAnnot	
	Degradation over Best	Improvement over Worst
compress	5.52% (378ms)	0.60% (80ms)
jess	9.22% (256ms)	38.87% (2220ms)
db	4.04% (745ms)	12.87% (2906ms)
javac	5.79% (380ms)	23.50% (2408ms)
mpegaudio	0.06% (1.2ms)	7.70% (518ms)
mtrt	12.71% (497ms)	17.40% (1421s)
jack	11.88% (476ms)	15.29% (865ms)
OptComp	-7.85% (-1388ms)	43.02% (10687ms)
JavaGrande	-7.19% (-5440ms)	17.84% (13916ms)
SPECjbb	3.63% (4.63 $10^6$ /tput)	16.03% (26.45 $10^6$ /tput)
Geo. Mean	3.56%	18.70%

**Table 4: Average performance differences (absolute error) between the GC switching system and the reference system for different heap sizes.**

difference between our switching framework making use of the automatic decision heuristic (AutoSwitch) and the reference system. The table uses the same format as we used previously for Table 4.

In some cases, e.g., mpegaudio, AutoSwitch performs better than GCAnnot. This is due to differences in the time at which the switching occurs. In both cases, the initial GC is GMS. For GCAnnot, we switch just prior to the start of execution; using AutoSwitch, we switch at the first GC at which heap residency remains as 60% following collection. For mpegaudio, GMS is more appropriate for its initial startup phase, and GC is more ap-

Average Difference Between Best & Worst GC Systems		
Benchmark	Auto Switch	
	Degradation over Best	Improvement over Worst
compress	7.84% (524ms)	0.97% (99ms)
jess	8.82% (245ms)	39.18% (2212ms)
db	12.80% (2358ms)	7.27% (1632ms)
javac	7.64% (484ms)	17.40% (2556ms)
mpegaudio	-13.83% (-849ms)	20.63% (1374ms)
mtrt	15.48% (594ms)	1.38% (163ms)
jack	12.62% (499ms)	10.32% (528ms)
OptComp	-11.89% (-1939ms)	45.80% (11376ms)
JavaGrande	5.60% (3406ms)	10.45% (81300ms)
SPECjbb	1.79% (2.29 10 <sup>6</sup> /tput)	18.41% (29.07 10 <sup>6</sup> /tput)
Geo. Mean	4.22%	16.37%

**Table 5: Average performance differences (absolute error) between our switching framework with *automatic switch decision support* and the reference system for different heap sizes.**

propriate during its steady state. As such, AutoSwitch outperforms GCAnnot. AutoSwitch achieves an average improvement of over 16% over the worst-performing GC system, across all benchmarks, and it imposes overhead that is under 5%. As an initial attempt, our simple heuristic performs quite well for the benchmark programs that we studied.

## 7. OTHER SWITCHING STUDIES

Two other studies demonstrate that switching collectors dynamically can be effective. In [50], the authors show that performance can be improved by combining variants of the same collector in a single system. They combine a mark-compact collector, and a mark-sweep non-compacting collector. They make use of the fast linear allocation of mark-compact, and the relatively fast mature GC time of mark-sweep in a generational context. Their heuristic uses mark-compact initially and also just after a heap expansion. When a full heap collection (major collection) is required, they switch to mark-sweep to take advantage of the fast old GC time. However, if linear allocation during promotion fails sufficiently, this implies that the old object space has become fragmented. Consequently, they switch to mark-compact to handle fragmentation. Their switching mechanism does not require reformatting the heap (they only need to create free-list structures).

In [54], the authors show that coupling compacting collectors with different performance characteristics can be effective. Semispace copying collection performs well when heap residency is low. However, mark-compact collection uses the heap space more efficiently. The authors combine semispace copying and mark-compact collectors to make use of these characteristics. When the heap residency is greater than a certain value, their system makes use of mark-compact collection, else it uses semispace collection. However, mark-compact collection is expensive if many objects survive collection, i.e., the basic heap residency is high. The authors propose a generalization of their system for generational collection. They also perform a limited evaluation of their non-generational switching system using a synthetic benchmark.

However, to our knowledge, no extant research has defined a general, easily extensible framework for switching between very diverse garbage collection systems, such as the one that we describe. In addition, our automatic switching heuristic, albeit simple, requires no user intervention and achieves considerable performance improvement.

## 8. CONCLUSION

Garbage collection plays an increasingly important role in next-generation Internet computing and server software technologies. In this paper, we first discussed some background for garbage collection, including basic garbage collection algorithms. However, basic GC algorithms are too simplistic to be successfully used for automatic memory management in virtual machines. We next surveyed modern garbage collection techniques that attempt to reduce automatic memory management overhead by making innovative use of basic garbage collection algorithms – we discussed modern techniques for maximizing application throughput, and techniques for minimizing pause time. In addition, we examined some supporting techniques that enhance performance.

We next discussed some garbage collection performance evaluation studies, which indicate that the performance of garbage collection techniques is largely dependent upon application execution behavior and resource availability. In addition, the overhead introduced by selection of the “wrong” collector can be significant. To overcome these limitations, we have developed a framework that can automatically switch between garbage collection systems without having to restart and possibly rebuild the execution environment, as is required by extant systems. Our system can switch between collection strategies *while* the program is executing. As such, it enables application-specific collection policies to be implemented that can also adapt to the underlying resource availability. The overhead introduced by our system is 4% for both annotation-guided and automatic switching using a simple heuristic, on average. Our system significantly improves performance (19% for annotation-guided and 16% for automatic switching) over the worst-performing collection system, on average.

We are currently working on ways to further reduce the overhead due to our framework, as mentioned in Section 6.1. We also plan to investigate other dynamic switching heuristics, in addition to the simple automatic switching technique that we detail in this work. These could be based on allocation rate, survival rate, and percentage of time spent in garbage collection. We are also considering other ways in which the virtual machine could adapt to resource availability, e.g., disabling certain optimizations when memory is constrained.

## 9. REFERENCES

- [1] ALPERN, B., ATTANASIO, C., BARTON, J., COCCHI, A., HUMMEL, S., LIEBER, D., NGO, T., MERGEN, M., SHEPHERD, J., AND SMITH, S. Implementing Jalapeño in Java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (Nov. 1999).
- [2] ALPERN, B., ET AL. The Jalapeño Virtual Machine. *IBM Systems Journal* 39, 1 (2000), 211–221.
- [3] APPEL, A. W. Simple generational garbage collection and fast allocation. *Software Practice and Experience* 19, 2 (1989), 171–183.
- [4] ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. Adaptive optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (Oct. 2000).
- [5] ATTANASIO, C. R., BACON, D. F., COCCHI, A., AND SMITH, S. A comparative evaluation of parallel garbage collectors. In *Fourteenth Annual Workshop on Languages and*

- Compilers for Parallel Computing* (Cumberland Falls, KT, Aug. 2001), Lecture Notes in Computer Science, Springer-Verlag.
- [6] AZATCHI, H., AND PETRANK, E. Integrating generations with advanced reference counting garbage collectors. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03)* (Warsaw, Poland, May 2003), vol. 2622 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 185–199.
- [7] BACON, D. F., ATTANASIO, C. R., LEE, H. B., RAJAN, V., AND SMITH, S. E. Java without the coffee breaks: A non-intrusive multiprocessor garbage collector. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Snowbird, Utah, Jun 2001).
- [8] BACON, D. F., FINK, S. J., AND GROVE, D. Space- and time-efficient implementation of the Java object model. In *Proceedings of the Sixteenth European Conference on Object-Oriented Programming* (Málaga, Spain, June 2002), B. Magnusson, Ed., vol. 2374 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 111–132.
- [9] BACON, D. F., AND RAJAN, V. Concurrent cycle collection in reference counted systems. In *Proceedings of 15th European Conference on Object-Oriented Programming, ECOOP 2001* (Budapest, June 2001), J. L. Knudsen, Ed., vol. 2072 of *Springer-Verlag*, Springer-Verlag.
- [10] BAKER, H. G. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices* 27, 3 (Mar. 1992), 66–70.
- [11] BARABASH, K., OSSIA, Y., AND PETRANK, E. Mostly concurrent garbage collection revisited. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2003), ACM Press, pp. 255–268.
- [12] BARTLETT, J. F. Compacting garbage collection with ambiguous roots. Tech. Rep. 88/2, DEC Western Research Laboratory, Palo Alto, CA, Feb. 1988. Also in *Lisp Pointers* 1, 6 (April–June 1988), 2–12.
- [13] BLACKBURN, S. M., JONES, R., MCKINLEY, K. S., AND MOSS, J. E. B. Beltway: getting around garbage collection gridlock. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (2002), ACM Press, pp. 153–164.
- [14] BLACKBURN, S. M., AND MCKINLEY, K. S. In or out?: putting write barriers in their place. In *Proceedings of the third international symposium on Memory management* (2002), ACM Press, pp. 175–184.
- [15] BLACKBURN, S. M., AND MCKINLEY, K. S. Ulterior referene counting: Fast garbage collection without a long wait. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications* (Anaheim, CA, Nov. 2003), ACM SIGPLAN Notices, ACM Press.
- [16] BLACKBURN, S. M., SINGHAI, S., HERTZ, M., MCKINLEY, K. S., AND MOSS, J. E. B. Pretenuing for java. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (2001), ACM Press, pp. 342–352.
- [17] BOEHM, H.-J., DEMERS, A. J., AND SHENKER, S. Mostly parallel garbage collection. *ACM SIGPLAN Notices* 26, 6 (1991), 157–164.
- [18] BOEHM, H.-J., AND WEISER, M. Garbage collection in an uncooperative environment. *Software Practice and Experience* 18, 9 (1988), 807–820.
- [19] BORMAN, S. Senible sanitation – understanding the IBM Java garbage collector (part 1: Object allocation). *IBM developerWorks* (Aug. 2002).
- [20] BOX, D. *Essential .NET, Volume I: The Common Language Runtime*. Addison Wesley Professional, Nov. 2002.
- [21] BRECHT, T., ARJOMANDI, E., LI, C., AND PHAM, H. Controlling garbage collection and heap growth to reduce the execution time of java applications. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (2001), ACM Press, pp. 353–366.
- [22] BURKE, M., CHOI, J., FINK, S., GROVE, D., HIND, M., SARKAR, V., SERRANO, M., SHREEDHAR, V., SRINIVASAN, H., AND WHALEY, J. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference* (June 1999), pp. 129–141.
- [23] CAUDILL, P. J., AND WIRFS-BROCK, A. A third-generation Smalltalk-80 implementation. In *Proceedings of the Conference on Object-Oriented Systems, Languages and Applications* (Oct. 1986), N. Meyrowitz, Ed., vol. 21(11) of *ACM SIGPLAN Notices*, ACM Press, pp. 119–130.
- [24] CHAMBERS, C. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for an Objected-Oriented Programming Language*. PhD thesis, Stanford University, Mar. 1992.
- [25] CHENEY, C. J. A non-recursive list compacting algorithm. *Communications of the ACM* 13, 11 (Nov. 1970), 677–8.
- [26] CHENG, P., HARPER, R., AND LEE, P. Generational stack collection and profile-driven pretenuing. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation* (Montreal, June 1998), ACM SIGPLAN Notices, ACM Press.
- [27] COHEN, J., AND NICOLAU, A. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems* 5, 4 (1983), 532–553.
- [28] DEUTSCH, L. P., AND BOBROW, D. G. An efficient incremental automatic garbage collector. *Communications of the ACM* 19, 9 (Sept. 1976), 522–526.
- [29] DIECKMAN, S., AND HÖLZLE, U. A study of the allocation behaviour of the SPECjvm98 Java benchmarks. In *Proceedings of 12th European Conference on Object-Oriented Programming, ECOOP98* (Brussels, July 1998), E. Jul, Ed., vol. 1445 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 92–115.
- [30] FINK, S. J., AND QIAN, F. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)* (San Francisco, California, March 2003).

- [31] FITZGERALD, R., KNOBLOCK, T. B., RUF, E., STEENSGAARD, B., AND TARDITI, D. Marmot: an optimizing compiler for Java. *Software – Practice and Experience* 30, 3, 199–232.
- [32] FITZGERALD, R., AND TARDITI, D. The case for profile-directed selection of garbage collectors. In *ISMM 2000 Proceedings of the Second International Symposium on Memory Management* (Minneapolis, MN, Oct. 2000), T. Hosking, Ed., vol. 36(1) of *ACM SIGPLAN Notices*, ACM Press.
- [33] HEWLETT-PACKARD COMPANY. NonStop Server for Java Software. Project home page. <http://nonstop.compaq.com/view.asp>.
- [34] HICKS, M., HORNOF, L., MOORE, J., AND NETTLES, S. A study of large object spaces. In *ISMM'98 Proceedings of the First International Symposium on Memory Management* (Vancouver, Oct. 1998), R. Jones, Ed., vol. 34(3) of *ACM SIGPLAN Notices*, ACM Press.
- [35] HÖLZLE, U. A fast write barrier for generational garbage collectors. In *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems* (Oct. 1993), E. Moss, P. R. Wilson, and B. Zorn, Eds.
- [36] HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)* (San Francisco, California, June 1992).
- [37] HOSKING, A. L., MOSS, J. E. B., AND STEFANOVIĆ, D. A comparative performance evaluation of write barrier implementations. In *OOPSLA'92 ACM Conference on Object-Oriented Systems, Languages and Applications* (Vancouver, British Columbia, Oct. 1992), A. Paepcke, Ed., vol. 27(10) of *ACM SIGPLAN Notices*, ACM Press.
- [38] HUDSON, R. L., AND MOSS, J. E. B. Incremental garbage collection for mature objects. In *Proceedings of International Workshop on Memory Management* (St Malo, France, 16–18 Sept. 1992), Y. Bekkers and J. Cohen, Eds., vol. 637 of *Lecture Notes in Computer Science*, Springer-Verlag.
- [39] HUDSON, R. L., MOSS, J. E. B., DIWAN, A., AND WEIGHT, C. F. A language-independent garbage collector toolkit. Tech. Rep. COINS 91-47, University of Massachusetts at Amherst, Department of Computer and Information Science, Sept. 1991.
- [40] IBM CORPORATION. WebSphere software platform. Product home page. <http://www-3.ibm.com/software/infol/websphere/index.jsp>.
- [41] INSTANTIATIONS INC. JOVE Optimizing Native Compiler for Java Technology. Project home page. <http://www.instantiations.com/jove/product/literature.htm>.
- [42] Java Grande Forum. <http://www.javagrande.org/>.
- [43] Java development kit version 1.2, summary of new features (performance enhancements). <http://java.sun.com/products/jdk/1.2/docs/relnotes/features.html>.
- [44] KRINTZ, C., AND CALDER, B. Using Annotation to Reduce Dynamic Optimization Time. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (June 2001), pp. 156–167.
- [45] LEA, D. A memory allocator, 1997. <http://gee.cs.oswego.edu/dl/html/malloc.html>.
- [46] LEVANONI, Y., AND PETRANK, E. An on-the-fly reference counting garbage collector for java. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (2001), ACM Press, pp. 367–380.
- [47] LIEBERMAN, H., AND HEWITT, C. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM* 26, 6 (1983), 419–429.
- [48] LINDHOLM, T., AND YELLIN, F. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.
- [49] MOON, D. A. Garbage collection in a large LISP system. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Austin, TX, Aug. 1984), G. L. Steele, Ed., ACM Press.
- [50] PRINTEZIS, T. Hot-Swapping between a Mark&Sweep and a Mark&Compact Garbage Collector in a Generational Environment. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)* (Monterey, CA, Apr. 2001).
- [51] PRINTEZIS, T., AND DETLEFS, D. A generational mostly-concurrent garbage collector. In *ISMM 2000 Proceedings of the Second International Symposium on Memory Management* (Minneapolis, MN, Oct. 2000), T. Hosking, Ed., vol. 36(1) of *ACM SIGPLAN Notices*, ACM Press.
- [52] ROSEN, M. BEA's enterprise platform. IDC white paper sponsored by BEA. <http://www.bea.com/framework.jsp>.
- [53] SACHINDRAN, N., ELIOT, J., AND MOSS, B. Mark-copy: Fast copying gc with less space overhead. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2003), ACM Press, pp. 326–343.
- [54] SANSOM, P. Combining single-space and two-space compacting garbage collectors. In *Proceedings of the 1991 Glasgow Workshop on Functional Programming* (Portree, Scotland, 1992), R. Heldal, C. K. Holst, and P. Wadler, Eds., Workshops in Computing, Springer-Verlag, pp. 312–323.
- [55] SMITH, F., AND MORRISETT, G. Comparing mostly-copying and mark-sweep conservative collection. In *Proceedings of the first international symposium on Memory management* (1998), ACM Press, pp. 68–78.
- [56] The SPECjbb2000 Java Business Benchmark. <http://www.specbench.org/osg/jbb2000/>.
- [57] SpecJVM'98 Benchmarks. <http://www.spec.org/osg/jvm98>.
- [58] Standard performance evaluation corporation (SpecJVM98 and SpecJBB Benchmarks). <http://www.spec.org/>.
- [59] STEFANOVIĆ, D., HERTZ, M., BLACKBURN, S. M., MCKINLEY, K. S., AND MOSS, J. E. B. Older-first garbage collection in practice: evaluation in a java virtual machine. In *Proceedings of the Workshop on Memory System Perfor-*

mance (2002), ACM Press, pp. 25–36.

- [60] STEFANOVIĆ, D., MCKINLEY, K. S., AND MOSS, J. E. B. Age-based garbage collection. In *OOPSLA '99 ACM Conference on Object-Oriented Systems, Languages and Applications* (Denver, CO, Oct. 1999), vol. 34(10) of *ACM SIGPLAN Notices*, ACM Press.
- [61] SUN MICROSYSTEMS. The Java HotSpot Performance Engine Architecture. Whitepaper. <http://java.sun.com/products/hotspot/whitepaper.html>.
- [62] UNGAR, D. Generation scavenging: A non-disruptive high performance storage recclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburg, Pennsylvania, Apr 1992).
- [63] WILSON, P. R. Uniprocessor garbage collection techniques. In *Proceedings of International Workshop on Memory Management* (16–18 Sept. 1992), vol. 637 of *Lecture Notes in Computer Science*.
- [64] WILSON, P. R., JOHNSTONE, M. S., NEELY, M., AND BOLES, D. Dynamic storage allocation: A survey and critical review. In *Proceeding of the International Workshop on Memory Management* (Kinross Scotland (UK), 1995).
- [65] WILSON, P. R., AND MOHER, T. G. A card-marking scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices* 24, 5 (1989), 87–92.
- [66] ZORN, B. Comparing mark-and sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM conference on LISP and Functional Programming* (1990), ACM Press, pp. 87–98.