

Application-Specific Garbage Collection^{1,2}

Sunil Soman Chandra Krintz

*Computer Science Department,
University of California, Santa Barbara, CA
{sunils,ckrintz}@cs.ucsb.edu*

Abstract

Prior work, including our own, shows that application performance in garbage collected languages is highly dependent upon the application behavior and on underlying resource availability. We show that given a wide range of diverse garbage collection (GC) algorithms, no single system performs best across programs and heap sizes.

We present a Java Virtual Machine extension for dynamic and automatic switching between diverse, widely used GCs for *application-specific garbage collection selection*. We describe annotation-guided, and automatic GC switching. We also describe a novel extension to extant on-stack replacement (OSR) mechanisms for aggressive GC specialization that is readily amenable to compiler optimization.

Key words: Application-specific garbage collection, dynamic GC selection, adaptive garbage collection, virtual machine, Java

1 Introduction

Managed runtime environments (MREs) are software systems that enable portable and secure execution of programs written in modern, high-level, programming languages such as Java and C#. One way we can ensure the continued success and widespread use of MREs (and these languages) is to improve the performance of MREs and the programs that they execute. One MRE mechanism that significantly impacts system and program performance is dynamic memory management.

MREs employ garbage collection (GC) to enable automatic memory reclamation and memory safety. GC consists of allocation and collection techniques that track and reclaim memory that is unreachable by a program at *runtime*. The GC must recycle heap memory effectively without imposing significant overhead on the executing program.

¹ This work was funded in part by NSF grants No. EHS-0209195 and No. ST-HEC-0444412, and an Intel/UCMicro research grant.

² This paper is an extension of the ISMM'04 paper titled Dynamic Selection of Application-specific Garbage Collectors by S. Soman, C. Krintz, and D. F. Bacon.

The performance of heap allocation and collection techniques has been the focus of much recent research (Blackburn et al., 2002; Brecht et al., 2001; Blackburn et al., 2001; Aiken and Gay, 1998; Hicks et al., 1998; Clinger and Hansen, 1997; Ungar and Jackson, 1992; Bacon et al., 2001; Hirzel et al., 2003). The goal of most of this prior work has been to provide general-purpose mechanisms that enable high performance execution across all applications. However, many researchers, including ourselves, find that the performance of a memory management system (the allocator and the garbage collector) is dependent upon application behavior and available resources (Soman et al., 2004; Attanasio et al., 2001; Fitzgerald and Tarditi, 2000; Zorn, 1990; Smith and Morrisett, 1998). That is, *no single collection system enables the best performance for all applications and all heap sizes* and the difference in performance can be significant. In our experimentation, we find that across a wide-range of heap sizes and the 11 benchmarks studied, each of *five different* collectors enabled the best performance at least once; this set of garbage collectors (Jones, 1996) includes those that implement semispace copying, generational collection, mark-sweep collection, and hybrids of these different systems. Prior work and these results indicate that to achieve the best performance, *the collection and allocation algorithms used should be specific to both application behavior and available resources*.

Furthermore, it is an open question whether current successful GC systems specialized for existing benchmarks, or specific environments (e.g., Sachindran et al. (2004); Hertz et al. (2005)), will be able to extract high performance from future applications that emerge and evolve. These next-generation applications will likely have very different behaviors and resource requirements than applications that exist today. A recent example of this evolution is the emergence of application servers that execute via MREs. These applications expose a very different profile than the client, GUI, and applet programs that came before them. MREs must be able to extract high-performance from emerging applications, i.e., adapt in a way that is application-specific.

Currently, MREs enable application- and heap-specific GC through the use of different configurations of the execution environment (via separate builds or command-line options). However, such systems require multiple images for different programs and preclude MRE persistence in which a single execution environment executes continuously while multiple applications and code components are uploaded by users (IBM Shiraz, 2001; IBM WebSphere, 2004; Hewlett-Packard NonStop, 2003; BEA Systems Inc., 2002; HALLoGRAM JRun, 2001). For extant MREs, 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 may result in significant performance degradation.

To address this limitation, yet to achieve application-specific GC, we present the design and implementation of a novel VM extension that is able to switch dynamically between a number of diverse GC systems at runtime. In this paper, we overview this system and describe how we are able to reduce the overhead of such an approach. In particular, we aggressively specialize the program for the underlying GC. Then, when a switch occurs, we “undo” the specializations so that we maintain correctness. To enable this, we employ method invalidation and on-stack-replacement (OSR) (Ungar and Smith, 1987; Hölzle et al., 1992; Fink and Qian, 2003).

Unfortunately, extant approaches to OSR are inefficient if the points at which OSR may occur (OSR points) are frequent. In our GC switching system for example, such points reside at every point in the code at which a garbage collection may be triggered (potential GC switch points). Existing OSR implementations restrict compiler optimization across OSR points and extend the live ranges of variables. As a result, they prevent the compiler from producing high-quality code, and thereby hinder efficient execution. We consider a novel approach to OSR and decouple its implementation from code generation. Our results indicate that this new version of OSR improves performance by 9% on average.

We also describe two techniques that use the efficient switching functionality that we provide: *annotation-guided GC selection* and *automatic GC switching*. For the former, we annotate each Java program with the best-performing GC (that we identify offline) for a range of heap sizes. Upon invocation, the JVM switches to the GC specified by the annotation for the currently available heap size. Our empirical evaluation shows that, on average, our system introduces around 4% overhead when it performs no switching. Annotation-guided switching improves performance by 21 – 34% (depending on heap size) over the worst-performing GC.

For automatic switching, our system dynamically monitors the performance of the GC and identifies when a switch will benefit performance. In particular, we describe a scenario in which memory resources are taken away from a JVM by the OS (e.g., for another, higher priority, application). Our system automatically switches to a GC that prevents OutOfMemory (Lindholm and Yellin, 1999) errors or improves performance given the newly available heap size. Our results indicate that our automatic switching system can improve performance by 22% on average for this scenario.

In the next section, we motivate application-specific garbage collection, followed by a description of our switching infrastructure in Section 3. We also describe our specialization and OSR techniques in this section. We then detail annotation-guided (Section 4), and automatic (Section 5) switching. We then evaluate the performance of our system with and without switching (Sec-

tion 6), present related work (Section 7) and conclude (Section 8).

2 Application-Specific Garbage Collection

To investigate the potential of application-specific GC, we first present experimental results for benchmark execution time using a wide-range of heap sizes, in Figure 1. This set of experiments confirms similar findings of others (Atanasio et al., 2001; Fitzgerald and Tarditi, 2000; Zorn, 1990) that indicate that no single GC system enables the best performance for all applications, on all hardware, and given all resource constraints.

The graphs show execution time over heap sizes with different garbage collectors for a few standard benchmarks – SPECjbb (SPEC Corporation, 1998), Voronoi from the JOlden benchmark suite (Cahoon and McKinley, 2001), and db from the SpecJVM98 suite (SPEC Corporation, 1998). We employ the widely used Jikes Research Virtual Machine (Jikes RVM) (Alpern et al., 2000) for our experimentation and prototype system. The x-axis is heap size relative to the minimum heap size that the application requires for complete execution across all GC systems. For SPECjbb, the y-axis is the inverse of the throughput reported by the benchmark; we report $10^6/\text{throughput}$ to maintain visual consistency with the execution time data of the other benchmarks. Lower values are better for all graphs.

The top-most graph in the figure shows that for SPECjbb, the semispace (SS) collector performs best for all heap sizes larger than 4 times the minimum, and the generational/mark-sweep hybrid (GMS) performs best for small heap sizes. The middle graph, for Voronoi shows that for heap sizes larger than 4 times the minimum, semispace (SS) performs best. For heap sizes between 2 and 4 times the minimum, mark-sweep (MS) performs best. Moreover, for small heap sizes (GMS) performs best. The bottom-most graph shows the performance of db: SS and GSS (a generational/semispace hybrid) perform best for large heap sizes, while CMS (a non-generational semispace copying/mark-sweep hybrid), and MS perform best for small heap sizes. The collectors will be described in detail shortly. These results support the findings of others (Atanasio et al., 2001; Fitzgerald and Tarditi, 2000; Zorn, 1990), that no single collection system enables the best performance across benchmarks. Further, no single system performs best *across heap sizes for a single benchmark/input pair*. We refer to any point at which the best-performing GC changes as a *switch point*.

To exploit this execution behavior that is specific to both the application and underlying resource availability, we extended Jikes RVM, to enable dynamic switching between GCs. The goal of our work is to enable application-specific garbage collection, to improve performance of applications for which there exist GC switch points, and to do so without imposing significant overhead. Such

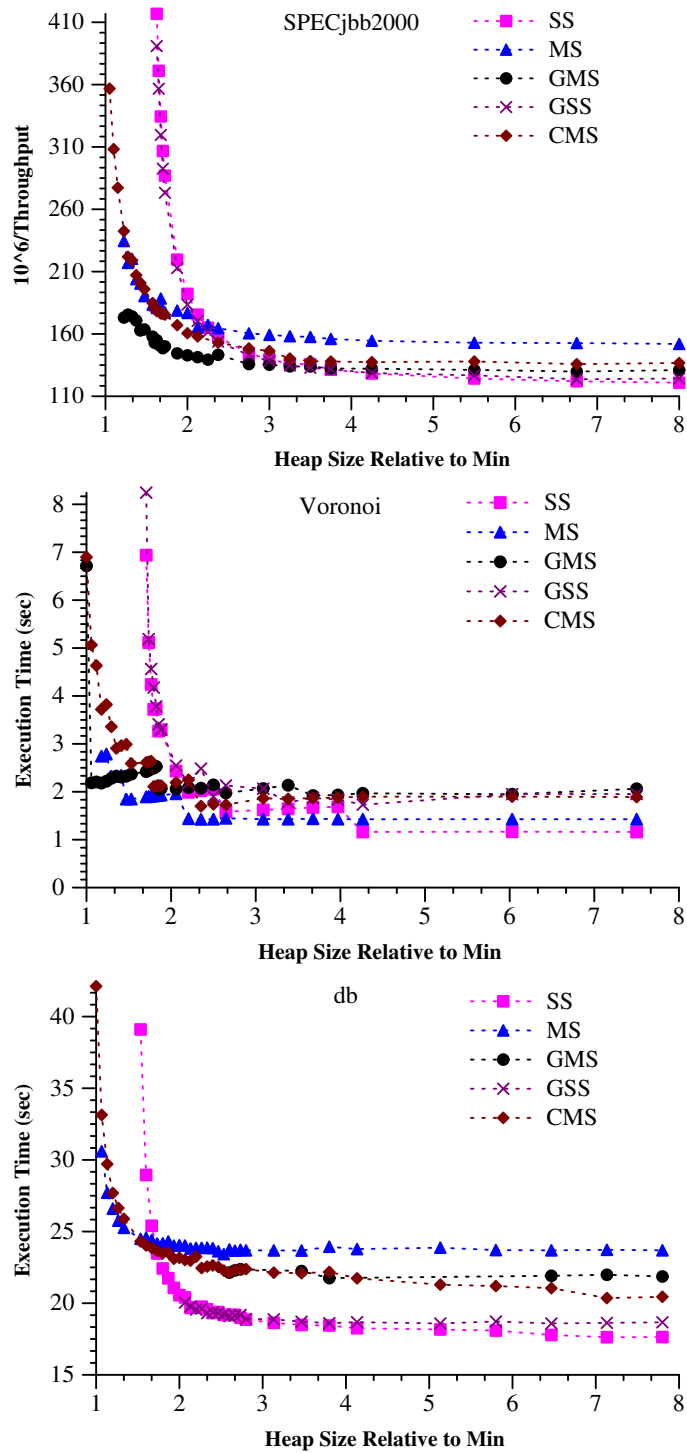


Fig. 1. Performance using different GCs and heap sizes in Jikes RVM. No single collector enables the best performance for all programs and heap sizes. The GCs that we used are Semispace copying (SS), Mark-sweep (MS), Generational/Mark-sweep (GMS) hybrid, Generational Semispace (GSS), and a non-generational Semispace/Mark-sweep collector (CMS). The y-axis is total time in seconds. For SPECjbb, we report $10^6/\text{throughput}$ to maintain visual consistency with the other graphs. The x-axis is heap size relative to the minimum in which the program can execute with the collector that can execute the program in the least heap size. The benchmarks are from the SPECjbb, Jolden, and SpecJVM98 suites.

a system will enable users to extract the best performance from their applications with such a MRE. Moreover, an MRE with GC switching functionality will be able to adapt to enable high-performance for future and emerging applications with little or no change to the MRE.

3 Support for Garbage Collection Switching

In this section, we describe various technical issues involved in enabling support for switching between garbage collectors at execution time.

Key to switching between collectors is efficient use of the available virtual address space between GCs. Different GC algorithms expect different heap layouts, such as a mark-sweep space, nursery, or a large object space. Virtual address space is limited and controls the maximum size of the heap. Hence, to make the best use of total available space, the virtual space needs to be divided carefully between different heap layouts.

In addition to virtual address space considerations, we need to support diverse object header information needed by copying and mark-sweep collectors. Copying and mark-sweep use the object header for different purposes, and support for both techniques involves enabling the use of state information used by both.

Another key concern is *specialization*. For performance, code is specialized for the current garbage collector. For instance, inlining of allocation sites, and the presence of write barriers based on whether or not the current garbage collector is generational. Since the garbage collector may change at runtime, assumptions made during compilation for specialization may change as well. Consequently, we need to be able to invalidate (recompile) specialized methods, and in addition, replace specialized code that is *executing* at the time of the switch. We shall describe mechanisms for invalidation in Section 3.3.

We discuss each of the above issues in detail below.

3.1 Multiple Garbage Collectors in a Single JVM

Jikes RVM (Alpern et al., 2000) is an open-source virtual machine for Java that employs dynamic and adaptive optimization with the goal of enabling high performance in server systems. Jikes RVM compiles Java bytecode programs at the method-level at runtime (just-in-time), to x86 (or Power PC) code. Jikes RVM supports extensive runtime services – garbage collection, thread scheduling, synchronization, etc. In addition, Jikes RVM implements adaptive or mixed-mode optimization by performing on-line instrumentation and profile collection, and then uses the profile data to evaluate when program characteristics have changed enough to warrant method-level re-optimization.

The current version of the Jikes RVM optimizing compiler applies three levels of optimization (0, 1 and 2). Level 0 optimizations include local propagation (of constants, types, copies), arithmetic simplification, and check elimination (of nulls, casts, array bounds). Moreover, as part of level 0 optimizations, write barriers are inlined into methods if the GC is generational. Level 1 optimizations include all of the level 0 optimizations as well as common subexpression elimination, redundant load elimination, global propagation, scalar replacement, and method inlining (including calls to the memory allocation routines). Level 2 includes SSA based optimizations in addition to level 1 optimizations.

Jikes RVM (version 2.2.0+) uses the Java Memory Management Toolkit (JMTk) (Blackburn et al., 2003) that enables garbage collection and allocation algorithms to be written and “plugged” into Jikes RVM. The framework offers a high-level, uniform interface to Jikes RVM that is implemented by all memory management routines. A GC is a combination of an allocation policy and a collection technique (this corresponds to a *Plan* in JMTk terminology). The JMTk provides the functionality that allows users to implement their own GC (without having to write one from scratch) and to perform an empirical comparison with other existing collectors and allocators. For this purpose, it provides users with utility routines for common GC operations, such as, copying, marking and sweeping objects. When a user builds a configuration of Jikes RVM, she is able to select a particular GC for incorporation into the Jikes RVM image.

The five GCs that we consider in this work are Semispace copying (SS), a Generational/Semispace Hybrid (GSS), a Generational/Mark-sweep Hybrid (GMS), a non-generational Semispace/ Mark-sweep Hybrid (CMS), and Mark-sweep (MS). These systems use stop-the-world collection and hence, require that all mutators be paused when garbage collection is in progress. Semispace copying and mark-sweep are standard non-generational collectors (Jones, 1996; Blackburn et al., 2003) with a single space for most mutator allocation (large objects are allocated in a separate space). Allocation in the semispace configuration is through a pointer increment (bump pointer), while that in mark-sweep involves a segregated free list. The free list is divided into several size classes and objects are allocated from the appropriate size class using a first-fit algorithm. Non-generational collectors collect the entire heap on every collection. Bump pointer allocation is believed to be much faster than free list allocation, since it is a much simpler operation.

The generational collectors, GSS and GMS, make use of well-known generational garbage collection techniques (Appel, 1989; Ungar, 1992). Young objects are allocated in an Appel-style (Appel, 1989) variable-sized nursery space using *bump pointer* (pointer increment) allocation from a contiguous block of memory. The boundary between the nursery and the mature space is dynamic. Initially, the nursery occupies half the heap, and the mature space is empty.

As live data from the nursery is promoted to the mature space on a minor collection, the size of the nursery shrinks. After a major (full heap) collection, the mature space contains live old data, and the nursery occupies half of the remaining space. Upon a minor collection, the nursery is collected, and the survivors are copied (promoted) to the mature space. Promotion is *en masse*, i.e., all survivors are copied to the mature space without first being moved to an intermediate space (Sun Microsystems, Inc., 2001). The mature space is collected by performing a full heap collection. This process is referred to as a *major collection*. Since minor collections are performed separately from major collections, pointers from mature space objects to nursery objects need to be identified to keep the corresponding nursery objects live. A *write barrier* is employed for this purpose. A write barrier is a series of instructions that is used to keep track of such mature space objects.

The main difference between GSS and GMS is the way in which the mature space is collected. GSS employs copying collection for this purpose, while GMS makes use of mark-sweep collection.

During a minor collection, nursery objects can be copied to the mature space in the GSS collector by a simple bump pointer allocation. However, allocation from the mature space in GMS is performed using a sequential, first-fit, free-list. GMS mature space collection is a two-phase process that consists of a mark phase in which live objects are marked, and a sweep phase in which unmarked space is returned to the free-list.

Generational GC performs well when the number of minor collections is large, since a minor collection is much faster than a full heap GC. However, when memory is plentiful, and GC is not required, non-generational collection may perform competitively. In fact, under these conditions, in several cases, semispace collection may commonly outperform other GCs due to cache locality benefits, and low fragmentation enabled by bump pointer allocation (Jones, 1996).

CMS³ is similar to a traditional semispace collector (SS) in that it is non-generational and is divided into two memory regions. However, CMS is a hybrid approach in which the first section is managed via bump pointer allocation and copy collection and the second section is managed via Mark-sweep collection (and uses free-list allocation as described above). CMS does not use write barriers. As a result, CMS is only able to identify references from the mark-sweep space to the semispace by tracing the objects in the former. Consequently, when a CMS collection occurs, the entire heap is collected – using copy collection for the first section then mark-sweep collection for the second section. CMS is more space efficient compared to a copying collector since it

³ This is a stop-the-world collector and should not be confused with the Concurrent Mark-Sweep collector in Sun's HotSpot VM (Sun Microsystems, Inc., 2001).

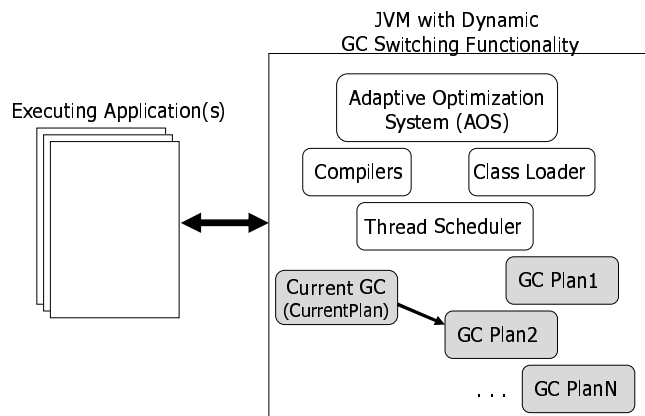


Fig. 2. Overview of our GC switching system. The JVM consists of the standard set of services as well as *multiple* GCs (an allocator and collector) as opposed to one per JVM image. The system employs the current GC through a reference to it called the CurrentPlan. When a switch occurs, the system updates the CurrentPlan pointer (and performs a collection in some cases). All future allocations and collections (if any) use the the newly selected GC.

does not require a copy reserve. It is supposed to achieve good performance when the number of objects promoted is low.

JVM system classes and large objects are handled specially in Jikes RVM/JMTk system. There is a separate immortal space that holds the Jikes RVM system classes. Allocation in the immortal space is via the bump pointer technique and this space is never collected. In addition, objects of size 16KB and greater are considered as “large objects”. The large object space is managed using mark-sweep collection. Collectors that employ a mark-sweep space, allocate large objects from the mature space (since these collectors already employ mark-sweep collection), while copying collectors employ a separate large object space.

Figure 2 shows the design of our GC switching system. Each VM image contains multiple GCs in addition to a set of standard services, such as the class loader, compilers, optimization system, and thread scheduler. Each GC consists of an implementation of an allocator and a collector.

The system switches to a new GC when doing so will improve performance. The system considers program annotations (if available), application behavior, and resource availability to decide when to switch dynamically, and to which GC it should switch. The GC currently in use is referred to by a CurrentPlan pointer. The compiler and runtime use this pointer to identify and employ the currently available GC. When a switch occurs, the system updates CurrentPlan to point to the new GC and performs allocation and collection (if needed) using the newly selected allocation and collection algorithms.

Each GC in Jikes RVM/JMTk is implemented as a *Plan*. The plan identifies

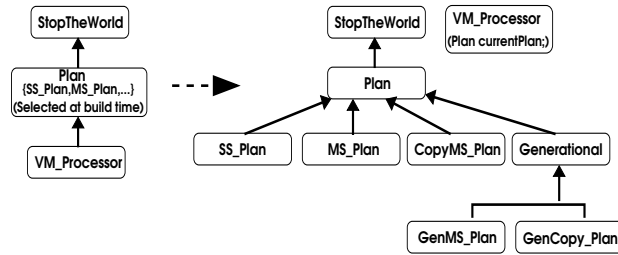


Fig. 3. Jikes RVM/JMTk class hierarchy: Original and switch-enabled.

the type of allocator and collector that is built into the image and consists of a set of classes that implement the appropriate algorithms for collection (semispace, generational, etc.) and allocation (free-list, bump pointer, etc.).

We extended Jikes RVM/JMTk system to implement each of Jikes RVM GCs within a single Jikes RVM image. We show the original and new Jikes RVM/JMTk class hierarchy in Figure 3. We implemented these classes so that much of the code base is reused across collection systems. The size of the VM image built with our extensions is 44.2MB (with the boot image compiled using the optimizing compiler at level 1), compared to an average size of 42.6MB for the reference Jikes RVM images (ranging from 37.2MB for SS to 49.4MB for MS) – our extensions do not significantly increase code size. Interestingly, the reference Jikes RVM image when built with MS, is larger than an image built with our modifications. We believe that the reason for this is that inlining of allocation sites for mark sweep increases code size significantly. Note that we do not inline allocation sites for boot image code in case of our switching system.

To support multiple GCs, we require address ranges for all possible virtual memory resources to be reserved. Our goal is to enable maximum overlap of virtual address spaces to reduce the overhead of switching. Our address space layout is shown in Figure 4(a). Each address range is lazily mapped to physical memory (as it is used by the executing program), in 1 Megabyte chunks. There are three shared spaces that we inherit from the default Jikes RVM implementation: the immortal (uncollected), GC Data Structure area (uncollected), and large object (>16KB) space. The GC that is currently in use employs a subset of other spaces as appropriate.

3.1.1 Switching Between GCs

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

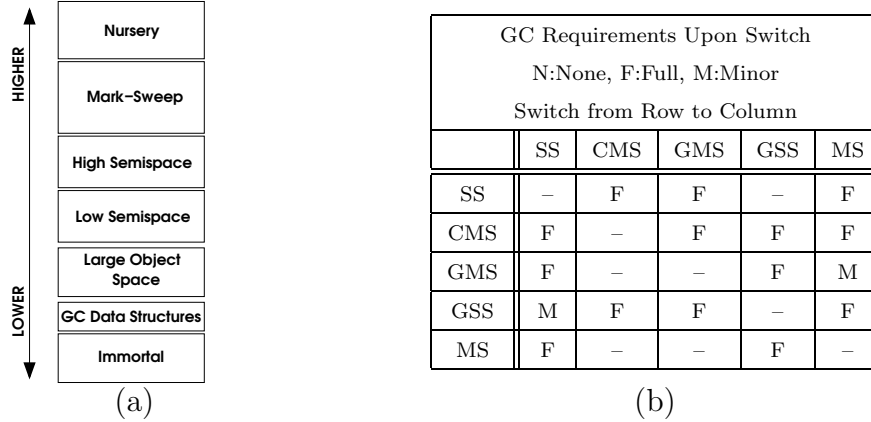


Fig. 4. Virtual address space layout in the switching system (a) and a table (b) that indicates when a GC is required on a switch (from the row GC to the column GC) and its type: full (F), minor (M), or none (-).

During a GC switch operation, we stop each executing mutator thread as if a garbage collection were taking place. A full heap GC, however, may not be necessary for all switches. To enable this, we carefully designed the layout of our heap spaces (Figure 4(a)) in such a way as to reduce the overhead of collection, i.e., to avoid a full garbage collection for as many different switches as possible. For example, a switch from SS to GSS only requires that future allocations by the application use the GSS nursery area since SS and GSS share their two half-spaces. Therefore, we only need to perform general bookkeeping to update the *CurrentPlan* to implement the switch.

Figure 4(b) indicates whether a GC is required, for a switch from the row GC to the column GC, and if it is, the type of GC required, e.g., full (F), minor (M), or none (N). We use the notation $XX \rightarrow YY$ to indicate a switch from collection system XX to collection system YY . The entries in the table show the type of GC that is required for row \rightarrow column. Note that we need to perform a garbage collection when switching from MS in only two cases (while switching to SS and GSS, the latter being a collector that is very often not the best choice, hence is not a frequent scenario). Moreover, MS commonly works well for very small heap sizes. We therefore use MS as our initial, default collector. As our system discovers when to switch to a more appropriate collection system, the cost of the switch itself is likely to be low.

We next describe the operations required for each type of switch. Whenever we perform a copy from one virtual memory resource to another, we use the allocation routine of the GC to which we are switching.

Switches That Do Not Require Collection. As mentioned above, $SS \rightarrow GSS$, $MS \rightarrow CMS$, $MS \rightarrow GMS$, and $CMS \rightarrow GMS$ do not require a collection since their virtual semispaces are shared.

Switches That Require Minor Collection. When we switch from a generational to a similar non-generational collector, e.g., $GMS \rightarrow MS$ and $GSS \rightarrow SS$, we need only perform a minor collection. That is, in addition to updating the *CurrentPlan*, we must collect the nursery space and copy the remaining live objects into the (shared) mature space.

Switches That Require Full Collection. The remaining switch combinations require a full garbage collection. We perform each switch as follows:

- **SS/GSS \rightarrow GMS/CMS/MS.** To switch between these collection systems, we perform a semispace collection (or a major collection for GSS). However, instead of copying survivors to the empty semispace, we copy them to the mark-sweep space of the target systems. When switching from GSS, we do the same; however, we must also copy the objects in the GSS mature space to the mark-sweep space.

Collectors that use semispaces (SS and GSS), require a copy reserve area, and consequently, do not perform well under memory pressure. In addition, if the ratio of live objects to dead is high, copying collectors involve expensive copying of live objects. Under such conditions, it would be beneficial to switch to a non-copying GC.

- **GMS/MS \rightarrow SS/GSS.** To perform this switch, we perform a major collection and copy survivors from the nursery and live objects from the mature space to the semispace. If we are switching from a non-generational MS system to SS or GSS, we mark live objects in the mark-sweep space and we forward them to the semispace resource. Since we must move objects during MS collection, we must maintain multiple states per object. We do this using an efficient, multi-purpose, object header described in Section 3.2.

If memory is plentiful, copying collectors can provide good performance since they employ fast, bump-pointer allocation. Also, certain applications might fragment the heap excessively, requiring compaction, which is inherently provided by copying collectors. Copying collection, is also supposed to provide better data cache locality, since objects are laid out in allocation order.

- **CMS \rightarrow Any GC.** Since there are no write barriers implemented for CMS, the heap spaces in this hybrid collector cannot be collected separately. Without write barriers to identify references from the mark-sweep space to the semispace, we may incorrectly collect live objects if we collect the semispace alone, i.e., those that are referenced by mark-sweep objects but not reachable from the root set. When we switch from CMS to any other GC, we must perform a full collection to ensure that we consider all live objects.

CMS is a compromise between generational, and non-generational collection. It does not incur the penalty of a write-barrier during mutation, yet provides segregation of old objects from young. However, CMS does not provide incremental behavior, i.e. the ability to collect only a part of the

heap (usually, the one with most likelihood of dead objects), independently of other parts, that generational collectors achieve.

Although the switching process is specific to the old and the new GCs, we provide an extensible framework that facilitates easy implementation of switching from any GC to any other, existing or future, that is supported by Jikes RVM JMTk. Moreover, unlike prior work, our system is able to switch dynamically between GCs that use very different allocation and collection strategies.

When a switch completes, we suspend the collector threads and resume the mutators, as is done during the post-processing of a normal collection. In addition, we *unmap* any memory regions that are no longer in use.

A limitation of the switching mechanisms described above is that we may not be able to perform certain kinds of switches when memory is highly constrained. For example, while switching from MS (or GMS, CMS) to SS (or GSS), we need to map the virtual address space corresponding to the SS *tospace*, on demand. However, we cannot unmap the MS address space until all live objects have been copied to the SS *tospace*. Consequently, our system requires more mapped memory than the reference system, *while* performing the switch in these cases. In practice however, switching from MS to SS or GSS when memory is constrained would be a poor choice (we provide further explanation of why this is the case in Section 4). A similar problem exists for switching from SS (or GSS) to a MS (or GMS, CMS) system. Note, however, that in these cases, we can unmap memory from the SS *tospace* before we copy objects to the MS space, since the SS *tospace* will not be used subsequently.

3.2 Multi-purpose Object Header

As mentioned in the previous section, to switch from a GC that uses a mark-sweep space (GMS, CMS, and MS) to a GC that uses a contiguous semispace (GSS, SS), we must maintain state for the mark-sweep process as well as for each object's forwarded location that is used by copying collection. Typically, garbage collectors store this state in the header of each object. In Jikes RVM, the garbage collectors each use a single 4-byte entry in the object header, called the *status word*.

The mark-sweep collector requires two bits in the status word: 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* enables efficient size-specific free-list allocation. Since the system aligns memory allocation requests on a 4-byte boundary, the lowest two 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 status word.

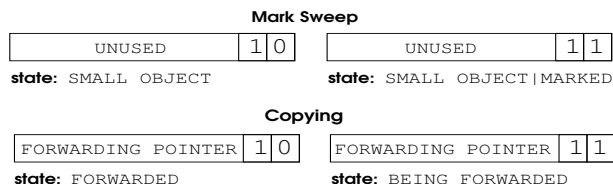


Fig. 5. Examples of bit positions in status word in object header

Semispace collectors also require header space to record the state of the copy process and the address to which the object is copied. A semispace collector marks an object as *being forwarded* while it is being copied. Once it is copied, the object is marked as *forwarded* and a forwarding pointer to the location to which the object was copied, is stored in the initial 30 bits of the header. The *being forwarded* state is necessary to ensure synchronization between multiple collector threads. These two states are stored in the two least-significant bits of the status word.

The two least-significant bits in an object status word implement different states depending on the collector. For example, as shown in Figure 5, if Jikes RVM is built using a mark-sweep GC, the value 0x2 in the two least-significant bits of the status word of an object indicates that the object is small and unmarked. 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 both bits are set, the status word indicates that the object is a small object and has been marked as live by a mark-sweep collector; the same state indicates to a semispace collector thread that the object is currently being forwarded by another thread.

Upon a switch from a collector that uses a mark-sweep space to one that uses a semispace, we must forward marked objects to the semispace. Consequently, our switching system must support *all four distinct states*, in addition to space for a forwarding pointer. To account for the two additional bits required and to avoid using an additional 4-byte header entry, we use bit-stealing (also used in prior GC systems (Bacon et al., 2002)) in which we “steal” the two least-significant bits from another address value that is byte-aligned.

The object header in Jikes RVM also stores a pointer to a Type Information Block (TIB) data structure, which provides access to the internal class representation and the virtual method table of the object. We use the two least-significant bits from the TIB pointer to store the additional states, *being forwarded* and *forwarded*, during the copying process. This implementation requires that we modify VM accesses to the TIB so that these bits are ignored. We found that this does not introduce significant overhead.

3.3 Specialization Support for GC Switching

A naïve switching implementation would involve two primary sources of overhead: write barriers are not needed by all collectors, and the loss of inlining opportunities due to dynamically changing allocation routines. Since our system can switch to a generational collector at any time, we would need to insert write barriers for every pointer field assignment in every method – these instructions would execute even when the collector in use is non-generational. Moreover, if the GC does not change over the lifetime of the program, we can inline calls to the allocation routine. However, in our system, the allocation routine may change, precluding our ability to inline.

To avoid a loss in performance due to these two issues, we *specialize the code* for the underlying GC aggressively and speculatively. That is, we inline allocation routines and insert write barriers only if the underlying GC is a generational collector.

For these specializations, we consider only optimized code. Jikes RVM, like many other commonly used JVMs (Sun Microsystems, Inc., 2001; Cierniak et al., 2000; Paleczny et al., 2001), employs adaptive optimization in which it only optimizes code that it identifies as hot, using efficient, online sampling of the executing program. Jikes RVM optimizing compiler applies three levels of optimization (0, 1, and 2) depending on how “hot” a method is. Level 0 optimizations include local propagation (of constants, types, copies), arithmetic simplification, and check elimination (of nulls, casts, array bounds). In addition, this level includes the inlining of write barriers into methods if the GC is generational. Level 1 optimizations include all level 0 optimizations as well as common subexpression elimination, redundant load elimination, global propagation, scalar replacement, and method inlining (including calls to allocation routines). Level 2 optimizations include all of level 1 optimization plus SSA-based transformations.

All unoptimized methods are compiled by Jikes RVM using a fast compiler that applies no optimization. We modified this compiler to insert write barriers into all methods regardless of the underlying collector. Since Jikes RVM itself is written in Java, all VM methods are compiled into a boot image – we modified this process as well to insert write barriers and to avoid inlining allocation routines into boot image methods. To enable speculative specialization, we modified level 0 of the optimizer so that it checked the `CurrentPlan` to determine whether to insert write barriers. We also modified level 1 (and above) to inline the allocation routines of the `CurrentPlan` collector. We made these changes in the runtime compiler (as opposed to the boot image compiler),

For annotation-guided GC selection, our system switches GCs immediately prior to the start of program execution. Therefore, no methods have been

optimized. Moreover, once the program begins, the system does not perform switching again. Thus, our specialization for write barriers and allocation routines is always correct in this case.

However, for automatic switching, the system can (and does) switch at any time. We therefore require a mechanism to “undo” the specializations when a switch occurs. We need only undo specializations that will cause incorrect execution. There are two such cases. First, the prior GC was not generational, the new GC is generational, and there is a field update in an optimized method. The new GC, therefore, requires a write-barrier for correctness. Second, there is an allocation site in an optimized method and the optimization level used by the compiler was 1 or higher. Consequently, the existing inlined allocation sequence is no longer valid and must be invalidated. For future invocations of these methods, we use method invalidation (Hölzle and Ungar, 1994) to undo the specialization. For methods that are currently executing, i.e., those that are on the runtime stack, we require on-stack-replacement (OSR) (Hölzle and Ungar, 1994; Chambers and Ungar, 1991; Hölzle et al., 1992; Fink and Qian, 2003; Paleczny et al., 2001; Sun Microsystems, Inc., 2001; Suganuma et al., 2003; Hölzle, 1994) of the method.

To enable OSR, the compiler must track the program execution state of the method at a particular program point in native code. The execution state consists of values for bytecode-level local variables, stack variables, and the current program counter. The execution state is a map that provides the OSR system with runtime values at the bytecode-level (source-level) so that the system can recompile and restart the method using another version. Existing OSR implementations insert a special (pseudo-) instruction, called an OSR point, to enable state collection.

OSR for replacement of executing optimized methods (as is needed for specialized methods in the GC switching system) is more complex than for unoptimized methods since compiler optimization can eliminate variables, combine multiple variables into one, and add variables (temporaries). This makes the ability to map bytecode-level variables correctly very challenging. All extant approaches to OSR avoid optimization across OSR points to avoid adding complexity to the compilation system. This, however, as we will later show, can significantly degrade code quality (and thus performance) if OSR support is to be enabled at a significant number of program points.

3.3.1 A Novel OSR Implementation

There are two reasons why extant approaches to OSR can degrade performance. First, all method variables (locals as well as stack) are considered live at an OSR point; by doing so, the compiler artificially extends the live ranges of variables and significantly limits the applicability of optimizations such as dead

code elimination, load/store elimination, alias analysis, and copy/constant propagation. Second, OSR points are “pinned” in the code to ensure that variable definitions are not moved around the OSR points; this precludes optimization and code motion across OSR points.

These prior implementations do not negatively impact performance (as a result of poor code quality) significantly when there are only a small number of OSR points. However, our switching system requires an OSR point at every point in the code at which a switch can occur; these are the points at which a GC can occur, i.e., gc-safe points. GC-safe points in Jikes RVM include implicit yield points (method prologues, method epilogues, and loop back-edges), call sites, exception throws, and explicit yieldpoints.

Since our GC switching system requires a very large number of OSR points, many along the critical path of the program, existing OSR implementations can severely degrade the performance of our GC switching system. We therefore extended Jikes RVM OSR implementation with a novel extension that is more amenable to optimization. In particular, we automatically track compiler optimizations in a specialized data structure to hold state information, called a variable map (VARMAP).

A VARMAP is a per-method list of bytecode variables (primitives as well as reference types) that are live at each gc-safe point. This list is independent of the code and does not impact the liveness information of the program point, nor does it restrict code motion optimizations. To ensure that we maintain accurate information in the VARMAP, we update it incrementally as compiler optimizations are performed. The VARMAP is somewhat similar in form to the data structure described in (Diwan et al., 1992), which was used to track pointer updates in the presence of compiler optimizations, for garbage collection support in Modula-3. However, unlike prior work, we track all stack, local, and temporary variables online, across a wide range of compiler optimizations automatically and transparently, during just-in-time compilation and dynamic optimization of Java programs.

Figure 6 shows an example of a VARMAP entry for a snippet of Java source. We include the equivalent Java bytecode and Jikes RVM high-level intermediate representation (HIR) of the code. Below the code, we show the VARMAP entry for the `callme()` call site. which contains the next bytecode index (25) after the call site `callme` and three typed local variables (`a: 18i`, `b: 115i`, `c: 117i`).

To update the VARMAP entries, we defined the following system methods:

- *transferVarForOsr(var1, var2)*: Record that `var2` will be used in place of `var1` from here on in the code (e.g., as a result of copy propagation)
- *removeVariableForOsr (var)*: Record that `var` is no longer live/valid in the

<pre> .. int c,d; b = a; c = b * 4; callme(); d = a + b; .. </pre>	<pre> .. 14: iload_1 15: istore_2 16: iload_2 17: iconst_4 18: imul 19: istore_3 20: invokestatic #3 //callme()V 23: iload_1 24: iload_2 25: iadd 26: istore_4 .. </pre>	<pre> .. 15: int_move 115i(int) = 18i(int) 18: int_shl 117i(int) = 115i(int), 2 20: call static "callme() V" 25: int_add 119i(int) = 18i(int), 115i(int) .. </pre>
		Intermediate Code (HIR)
		<pre> 25@main(..LLL,..),.., 118i(int), 115i(int), 117i(int), .. bcindex: 25, L: local var, a: 18i, b: 115i, c: 117i </pre>
Source	Byte Code	VARMAP entry

Fig. 6. Shows how the VARMAP is maintained for a snippet of Java source (its bytecode and high-level intermediate representation (HIR) is included). We show the VARMAP entry for the `callme()` call site which contains the next bytecode index (25) after the call site `callme` and three local variables with types (**a**: 18i, **b**: 115i, **c**: 117i).

Intermediate Code (HIR)	<pre> .. 15: int_move 115i(int)=18i(int) 18: int_shl 117i(int)=115i(int), 2 20: call static "callme() V" 25: int_add 119i(int)= 18i(int), 115i(int) .. </pre>	<pre> .. 15: int_move 115i(int)=18i(int) 18: int_shl 117i(int)=18i(int), 2 20: call static "callme() V" 25: int_add 119i(int) = 18i(int), 18i(int) .. </pre>
VARMAP entry	<pre> 25@main(..LLL,..),.., 118i(int), 115i(int), 117i(int), .. </pre> <p style="text-align: center;">transferVarForOsr(115i, 18i)</p>	<pre> 25@main(..LLL,..),.., 118i(int), 18i(int), 117i(int), .. </pre>
	Before optimization	After optimization

Fig. 7. Shows how the VARMAP is updated after copy propagation. Variable **b**: 115i is replaced with **a**: 18i.

code. Note that, even though a variable may not be live, we must still remember its relative order among the set of method variables.

- *replaceVarWithExpression(var, vars[], operators[])*: Record that variable **var** has been replaced by an expression that is derivable from the set of variables **vars** and **operators**.

Our OSR-enabled compilation system handles all extant Jikes RVM optimizations at all optimization levels. Each time a variable is updated by the compiler, the update occurs through a wrapper function that automatically invokes the necessary VARMAP functions. This enables us to easily extend the compilation system with new optimizations that automatically update the VARMAP appropriately. For example, for copy and constant propagation and CSE (common sub-expression elimination), when a use of a variable is replaced

by another variable (or constant), the wrapper function performs the replacement in the VARMAP record by invoking the `transferVarForOsr` function as shown in Figure 7 for an update that results from copy propagation.

We also update the VARMAP during live variable analysis. We record variables that are no longer live at each potential OSR point, and record the relative position of each in the map. We set every variable that live-analysis discovers as dead, to a `void type` in the VARMAP. We identify local and stack variables by their relative positions in the Java bytecode. Maintaining the relative positions of variables in the VARMAP allows us to restore a variable’s runtime value to the correct variable location.

During register allocation, we update the VARMAP with the actual register and spill locations for the variables, so that they can be restored from these locations during on-stack replacement. The VARMAP contains symbolic registers corresponding to each variable. We update symbolic registers with a physical register or a stack location upon allocation by querying the map maintained by the register allocator for every symbolic register that has been allocated to a physical register. We record spilled variables via the spill location that the allocator encodes as a field in the symbolic register object.

When the compilation of a method completes, we encode the VARMAP of the method using the compact encoding implemented for OSR points in the original system (Fink and Qian, 2003). The encoded map contains an entry for each potential OSR point. Each entry consists of the `register map`, which is a bit map that indicates which physical registers contain references (which a copying garbage collector may update). In addition, the map contains the current program counter (bytecode index), and a list of pairs (`local variable`, `location`) (each pair encoded as two integers), for every inlined method (in case of an inlined call sequence). The encoded map remains in the system throughout the lifetime of the program and all other data structures required for OSR-aware compilation (including the original VARMAP) are reclaimed during GC.

3.3.2 Triggering On-Stack Replacement

During execution, following a GC switch, we trigger OSR lazily as is done in Self for debugging optimized code (Hölzle et al., 1992). We tag a specialized method at compile time, and read this tag during GC switch to identify the method as specialized. We modify the return address of the specialized method’s callee so that it will jump to a special utility method that performs OSR for the specialized method. By triggering OSR lazily, we eliminate the need for runtime checks in the application code.

The utility method extracts the execution state from the stack frame of the specialized method, and sets up the new stack frame. To preserve register values contained in registers for the execution of specialized methods, the

helper saves all registers (volatiles and non-volatiles) into its stack frame. Since the helper is not directly called from the specialized code, we must “fake” a call to the helper. This involves setting the return address of the helper to point to the current instruction pointer in the specialized code upon entry to the helper. This process also requires that we update the stack pointer for the helper appropriately.

In the next section, we describe two uses of the framework for garbage collection switching – *annotation-guided switching*, and *automatic switching*.

4 Annotation-Based Garbage Collector Selection

By implementing the functionality to switch between collection systems while Jikes RVM is executing, we can now select the “best-performing” collection system for each application that executes using our system. To this end, we implemented *Annotation-guided GC System Selection*. In particular, we use a class file annotation to identify per-application garbage collectors that should be employed by our GC Switching system. We compactly encode the annotation in a class file that contains a `main(...)` method using a technique that we developed in prior work (Krintz and Calder, 2001).

To identify the GC that we recommend as an annotation, we analyzed application performance offline using the different Jikes RVM GCs. We considered a number of different heap sizes and program inputs. We list the inputs in Figure 9 and refer to them as Input and Cross. We extracted, for each heap size, the best performing GC across inputs. In addition, for benchmarks for which there were multiple best-performing GCs for different heap sizes, we also identified the *switch points* for each program, i.e., the heap sizes at which the best-performing GC changes.

For all benchmarks that we studied, the per-GC performance was very similar across inputs. Only one benchmark exhibited differences in the best-performing GC across inputs (JavaGrande). All other benchmarks showed no change in the choice of the GC across the inputs that we used. To investigate this further, we looked at several inputs for the SPECjbb benchmark, which is an example of a GC-intensive server program. For 4 different inputs for SPECjbb, we found that GMS enables best performance for small or medium heaps, while SS works best for large heaps (see Figure 8). This *input independence* appears to be very different from other types of profiles, such as, method invocation counts, field accesses, etc., in which cross-input behavior can vary widely (Krintz, 2003; Krintz and Calder, 2001). Therefore, we believe that it is less likely that we will negatively impact performance for inputs that we have not profiled. To select the GC to provide as an annotation for JavaGrande, we identified the GC that imposed the smallest percent degradation over the best performing collector across inputs at a range of heap sizes.

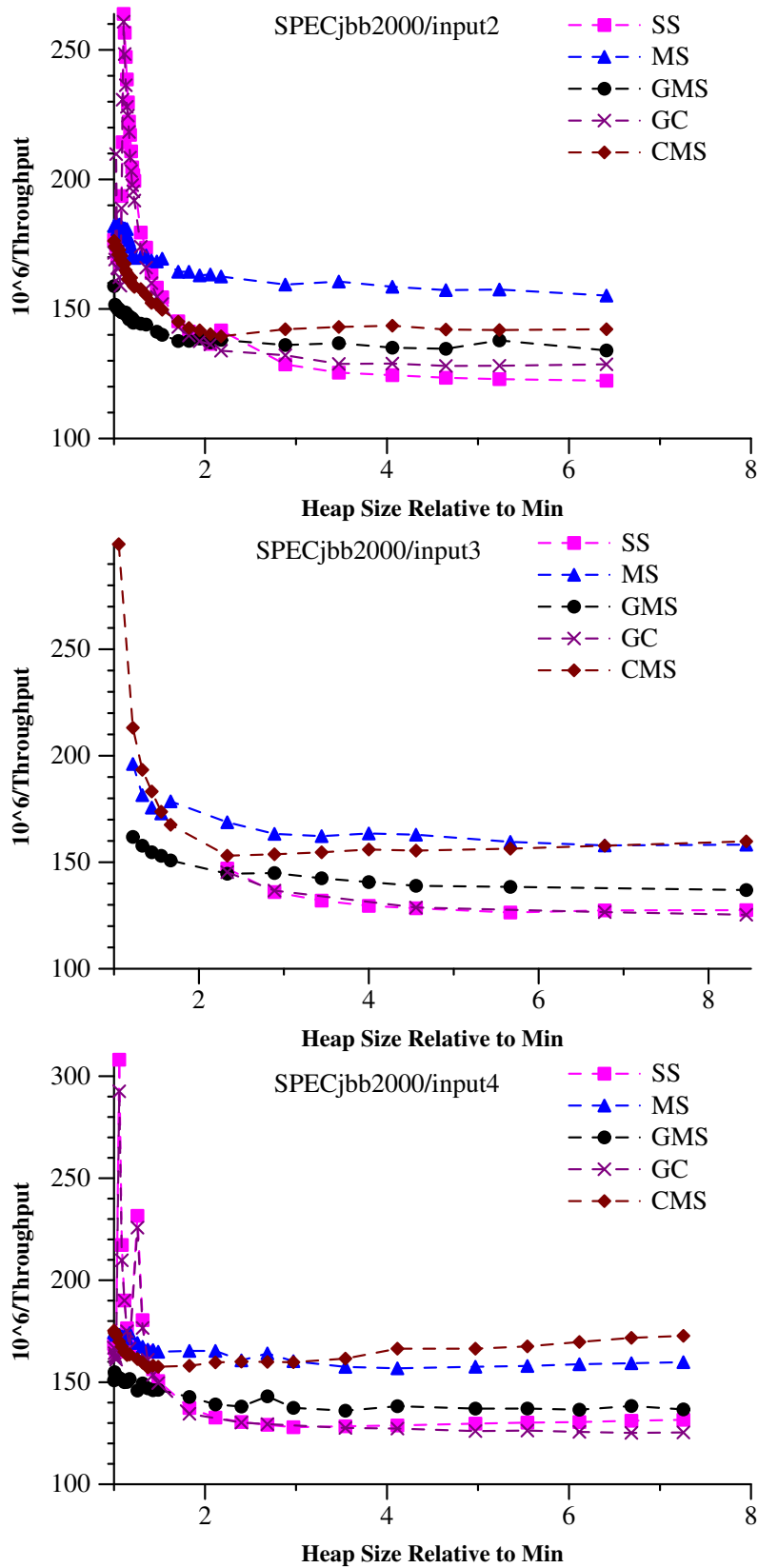


Fig. 8. The figure shows 3 different inputs for SPECjbb2000 (in addition to input1 in Figure 1). The x-axis is heap size relative to the minimum (1 to 8 times the minimum). The y-axis is the inverse of the throughput; we report $10^6/\text{throughput}$ to maintain visual consistency with the execution time data in the rest of the figures.

Benchmark	Input/Cross	Min Heap (MB)	Annot GC Selector	
			GC(s)	Switch Ratio
compress	100/10	21	SS	—
jess	100/10	9	GMS	—
db	100/10	15	CMS/SS	1.73
javac	100/10	30	GMS	—
mtrt	100/10	16	GMS	—
jack	100/10	18	GMS	—
JavaGrande	AllSizeA/SizeB	15	GMS/SS	3.00
MST	1050 nodes/640	78	MS/CMS	1.47
SPECjbb2000	1 warehouse/2	40	GMS/SS	3.00
Voronoi	65000 pts/20000	34	MS/SS	4.26

Fig. 9. This table shows the inputs that we considered to evaluate GC behavior across heap sizes, the minimum heap size in which the program will run using our JVM, and the GC selection decisions with which we annotate each program to enable annotation-guided GC switching.

The values that we annotate are shown in the final two columns of Figure 9. For each benchmark, we specify the GC that performs best. If there is more than one best-performing GC for different heap sizes, i.e., there is a *switch point*, we annotate each of the GCs and the switch point.

We found that for all of the benchmarks studied, if there was a switch point, there was only a single switch point and that the switch point heap size was very similar relative to the minimum heap size for each input. As such, we specify the switch point as the *ratio* of switch point heap size and the minimum heap size.

At program load time, the JVM computes the ratio, $\frac{\text{current_max_heap_size}}{\text{min_heap_size}}$, and compares this value with the annotated ratio. If the computed ratio is less, the JVM switches to the first GC, or to the second GC, otherwise. This requires that we also annotate the minimum heap size for the program and input. By doing so, we reduce the amount of offline profiling required by users of our system since, given the minimum heap size for an input, we can compute the switch point using the ratio from any input. We found that the switch point ratio holds across inputs for all of the benchmarks that we studied. Five of the eleven programs have switch points.

5 Automatic Garbage Collector Switching

In addition to annotation-guided GC, we investigated a mechanism to guide switching decisions automatically, when resources are suddenly constrained. In this scenario, the operating system (OS) reclaims virtual memory from our JVM for allocation and use by another process. Such a scenario is common to

server systems that execute many competing tasks concurrently.

The scenario that we investigated was one in which the program executes using a sufficiently large heap size, e.g., 200MB. During execution the OS reclaims memory and thereby reduces memory that is available for the heap in use by the executing program. In some cases, this may cause an OutOfMemory error when there is not sufficient virtual memory for the program to make progress.

Our switching system has an advantage in these cases since it can switch to a GC that makes more efficient use of the heap when resources are constrained, e.g., a non-copying system vs. a copying collector. We can switch to such a GC and allow the program to make progress and to avoid termination via the OutOfMemory error. In addition, by switching to a system that performs better under restricted resources, we can reduce the number of garbage collections that are performed, which may improve performance.

We employ a set of heuristics to determine when to switch. The GC switching system monitors the time spent in GC versus the time spent in the application threads. When this *GC load* exceeds 1 for an extended period of time, the system switches to a GC that is more appropriate when resources are constrained. In addition, we also switch GCs when we find that garbage collections are being triggered too frequently, measured as the duration for which the application threads execute between successive garbage collections. We initially use semispace copying collection (SS) when the program starts (as opposed to MS for the annotation-driven case). SS is the best performing collector across the programs that we studied – when the heap size is large (>200MB). When the switch occurs, the system employs Generational/Mark-Sweep (GMS); GMS performs best when resources are constrained. GMS has more available mature space (since it is mark-sweep collected) compared to other generational collectors. GMS performs no copying for the mature space and thus, when the GCs are frequent, less overhead is imposed on the program, unlike a copying collector.

We evaluate these heuristics and scenarios in our evaluation section. Though this set of heuristics is simple, we show that the GC switching functionality can achieve significant performance benefits (as well as avoid OutOfMemory errors). We plan to investigate other opportunities for automatic GC switching, e.g., to improve locality given changes in program phase behavior, as part of future work.

6 Evaluation

To empirically evaluate the effectiveness of switching between garbage collectors dynamically, we performed a series of experiments using our system and a number of benchmark programs. We first describe these benchmarks and

our experimental methodology with which we generated the results.

6.1 *Experimental Methodology*

We gathered our results using a dedicated 2.4GHz x86-based Xeon machine (with hyperthreading enabled) running Debian Linux v2.4.18. We implemented our switching framework within Jikes RVM version 2.2.0 with IBM jlibraries (Java libraries) R-2002-11-21-19-57-19. We employ a pseudo-adaptive Jikes RVM configuration (Sachindran and Moss, 2003) in which we capture the methods that Jikes RVM identifies as hot in an offline, profiled run. We then optimize those methods when they are first invoked to avoid the Jikes RVM learning time (Krintz, 2003), to reduce the non-determinism inherent in the adaptive configuration, and to enable the repeatability of our results. The boot image is compiled using the optimizing compiler (level 1).

We measured the impact of switching on application performance separately from compilation overhead. To enable the former, we executed the benchmarks through a harness program. The harness repeatedly executes the programs; the first run includes program compilation and later runs do not since all methods have been compiled following the initial invocation. We report results as the average of the wall clock time of the final 5 of 10 runs through the harness. We experimented with a range of programs from various benchmark suites, e.g., SpecJVM98 and SPECjbb (SPEC Corporation, 1998), JOlden (Cahoon and McKinley, 2001), and JavaGrande (Java Grande Forum, 1998) – we omit mpegaudio from the SpecJVM suite, since it exhibits very little allocation behavior and does not exercise memory extensively.

6.2 *Results*

We next present the empirical evaluation of our system. We first evaluate the impact of our new, VARMAP-based OSR implementation when we do not switch. We then evaluate the performance of annotation-guided and automatic GC switching.

VARMAP-Based OSR Performance

We first present results that compare our VARMAP-based OSR implementation to a variation of a commonly used, extant approach to OSR. To implement the latter, we employed the original OSR implementation in Jikes RVM. This implementation uses special, unconditional, OSR point instructions to allow OSR at a particular point in the execution. This implementation is used for deferred compilation and method promotion in the original system (Fink and Qian, 2003). We insert OSR points at each gc-safe point (all points at which a GC switch can occur) in each optimized method. An OSR point is a special thread yield point that will trigger on-stack replacement, unconditionally, for the current method. We remove these instructions immediately prior to code

generation (after all optimizations) to avoid their execution, since doing so will trigger OSR. By doing so, we are able to measure the impact of OSR on code quality alone.

Figure 10(a) shows the results from this comparison. The y-axis is the percent reduction in execution time enabled by OSR over OSR points (when OSR points are inserted at every gc-safe point during compilation as described above). The Average bar shows the average across all benchmarks, and Average Spec98 shows the average for only the SpecJVM benchmarks. We gathered results for 25 different heap sizes from the minimum in which the application would run to 8x the minimum at periodic intervals. We report the average over these heap sizes for each benchmark.

Our VARMAP implementation improves overall application execution time by 9% on average across all benchmarks, and by over 10% on average across the SpecJVM benchmarks. `jess` and `mtrt` show the most benefit, with improvements of 31% and 20% respectively. For these benchmarks, the original system increases register pressure by extending live ranges of variables. This results in a large number of variable spills to memory. Since we maintain the VARMAP separately from the compiled code, we ensure that live ranges are dictated by the code itself.

Figure 10(b) details the space and compilation overhead of our OSR implementation. Columns 2 and 3 show the compilation time for the clean (reference) Jikes RVM system without OSR points and the VARMAP implementation, respectively. Column 4 shows the percentage degradation in compilation time imposed by our VARMAP implementation. Columns 5 and 6 show the space overhead introduced by the VARMAP implementation during compile time (collectable) and runtime (persistent), respectively. On average, our system increases compile time by approximately 26% and adds 132KB of collectable overhead and 30KB of constant space overhead.

We next present results that show the overhead of our VARMAP implementation in our GC switching system when it *never switches* compared to the clean or reference Jikes RVM. This is to enable us to evaluate the effectiveness of our VARMAP in reducing the base overhead of the switching system, introduced due to loss of optimization opportunities. The switching system adds an overhead of around 15% on average across applications, when switching is never triggered (see Figure 11).

Figure 12 shows these results. The numbers show the percentage degradation introduced by the GC switching with the VARMAP implementation (without switching) over the reference Jikes RVM image across all measured heap sizes (minimum for each application to large). Average is the average percentage degradation across all benchmarks (5%), and Average Spec98 is the

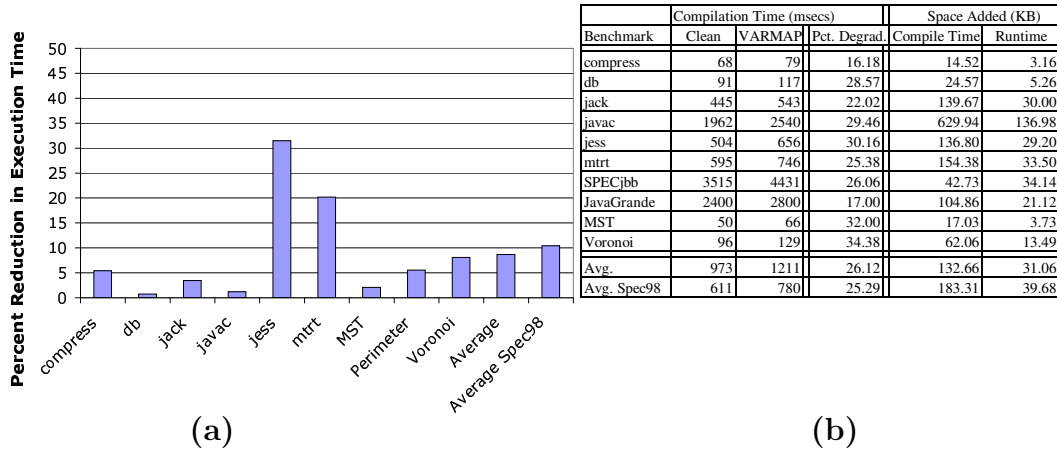


Fig. 10. Performance of our OSR-VARMAP Implementation in Jikes RVM Reference System. Graph (a) shows the average execution time (excluding compilation) performance improvement enabled across heap sizes by our VARMAP implementation over using an extant implementation of OSR – a variation on the OSR points in Jikes RVM. The table in (b) shows the compilation overhead of our VARMAP implementation over the reference system. Columns 2 and 3 are compilation times in milliseconds and column 4 is the percent increase in compilation time. The final two columns show the compilation (collectable) and runtime space overhead, respectively, introduced by our system.

Benchmark	Pct. Degradation No switching vs. Clean
compress	8.98
jess	29.78
db	3.12
javac	12.87
mpegaudio	24.04
mtrt	25.33
jack	6.83
Average	15.85

Fig. 11. Table shows the overhead introduced by the garbage collection switching system when it never switches, over the clean (reference) JikesRVM. The percentage values are averaged over heap sizes. On average, the GC switching system adds a 15% overhead over the clean JikesRVM, *when no switching is triggered*, due to support for on-stack replacement.

average percent degradation for only the Spec98 benchmarks (<5%). javac has a higher overhead (11%) than other benchmarks due to a larger space (and hence GC) overhead required to store the VARMAP information (see table (b) in Figure 10).

Annotation-Guided GC Selection

To investigate the effectiveness of our GC switching system, we implemented and evaluated annotation-guided and automatic GC selection. In this section, we present results for the former. As we described in Section 4, we selected the best-performing GC for a range of heap sizes by profiling multiple inputs

Benchmark	Pct. Degredation Over Clean
compress	3.71 (285ms)
db	3.09 (662ms)
jack	5.88 (269ms)
javac	11.31 (898ms)
jess	3.06 (104ms)
mtrt	0.62 (81ms)
SPECjbb	3.99 (5908ms)
JavaGrande	3.01 (1944ms)
MST	9.99 (237ms)
Voronoi	5.42 (245ms)
Average	5.01 (1063ms)
Average Spec98	4.62 (383ms)

Fig. 12. The overhead introduced by the VARMAP version of the GC Switching System over a clean system without GC switching functionality. By reducing the overhead of the Orig-OSR implementation, we are able to cut the base overhead of the GC switching system (the overhead imposed when the system *does not switch*) from 15% to 5%, i.e. the resulting version of the system introduces 5% base overhead over the clean system.

offline (we list the inputs in Figure 9). The GCs and switch points that we annotate and use are shown in the same table. For brevity, we present results only for the large input.

Our system uses the annotation to switch GCs immediately prior to invocation of the benchmark (at program load time). Our performance numbers *include* the cost of this switch. Moreover, we *specialize* the code for the underlying GC. Our system compiles hot methods with the appropriate allocation routine inlined. In addition, we insert write barriers into all unoptimized (baseline compiled) methods; however, write barriers are inserted into optimized (“hot”) methods for generational collection systems. Since our system switches to the annotated GC before the benchmark begins executing, no invalidation or on-stack replacement is required for annotation-guided switching.

As we discussed in Section 4, half of the benchmarks that exhibit a switch point. Given such benchmarks and our system’s ability to switch between GCs given the maximum available heap size, our system has the potential to enable significant performance improvements since no single collector is the best-performing across heap sizes for these programs even for the *same* input.

Figures 13 and 14 present performance graphs for representative benchmarks for a range of different heap sizes (x-axis – values are relative to the minimum heap size of the program). The y-axis is program execution time in seconds. For SPECjbb, the y-axis is the inverse of the throughput multiplied by 10^6 ; we report this metric to maintain visual consistency with the execution time data,

i.e., lower numbers are better. The y-axis value ranges vary across benchmarks.

Each graph contains six curves, one for each of the Jikes RVM garbage collectors. These curves represent the performance of the standard JikesRVM garbage collectors in the “clean”, unmodified, system, in addition to our GC annotation system. The GCs that we evaluate include Semispace (SS), a Generational/Semispace Hybrid (GSS), a Generational/Mark-sweep Hybrid (GMS), a non-generational Semispace/Mark-sweep Hybrid (CMS), and Mark-sweep (MS). The *GC Annot* curve (dashed line with + markers, red if in color) shows the performance of our GC switching system using annotation-guided selection.

The first set of graphs are three representative benchmarks that have switch points (those that exhibit a change in best-performing GC). Our system is able to track the best-performing GC for both small and large heap sizes. For example, for db, our system tracks CMS for small heaps and SS for large heaps. As such, for a *single program and input* but different resource availability levels, we can improve performance over using *any single collector* for these programs.

The second set shows three representative benchmarks without switch points. For these benchmarks, our system tracks the best-performing collector. Notice that the best-performing collector differs across programs, e.g., SS performs best for compress and GMS performs best for the others. Since our system uses annotation to guide GC selection and switch dynamically to the best-performing GC for each program, it is able to improve performance across benchmarks over any single GC. This becomes more evident when we evaluate this data across benchmarks.

Figure 15 and 16 summarize our results across benchmarks and heap sizes. Figure 15 represents averages for small heaps (minimum for an application to 3x the minimum), and Figure 16 represents averages for medium to large heap sizes (from 3x the minimum for an application to 8x the minimum heap size). We present the average difference between our GC switching system and the best-performing GC at each heap size (column 2) and between our system and the worst-performing GC 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 of the throughput. The tables shows that our system improves performance by 34% over selection of the “wrong”, i.e., worst-performing collector, for small heaps, and by 21% for medium to large heaps. In addition, the data shows the average performance degradation over optimal selection. This degradation is due to the implementation differences in our system that make it flexible, e.g., write barrier execution in unoptimized code, boot image optimization, switch time (from MS, the default system, to the annotated system), etc. On average, our system imposes a 4% overhead

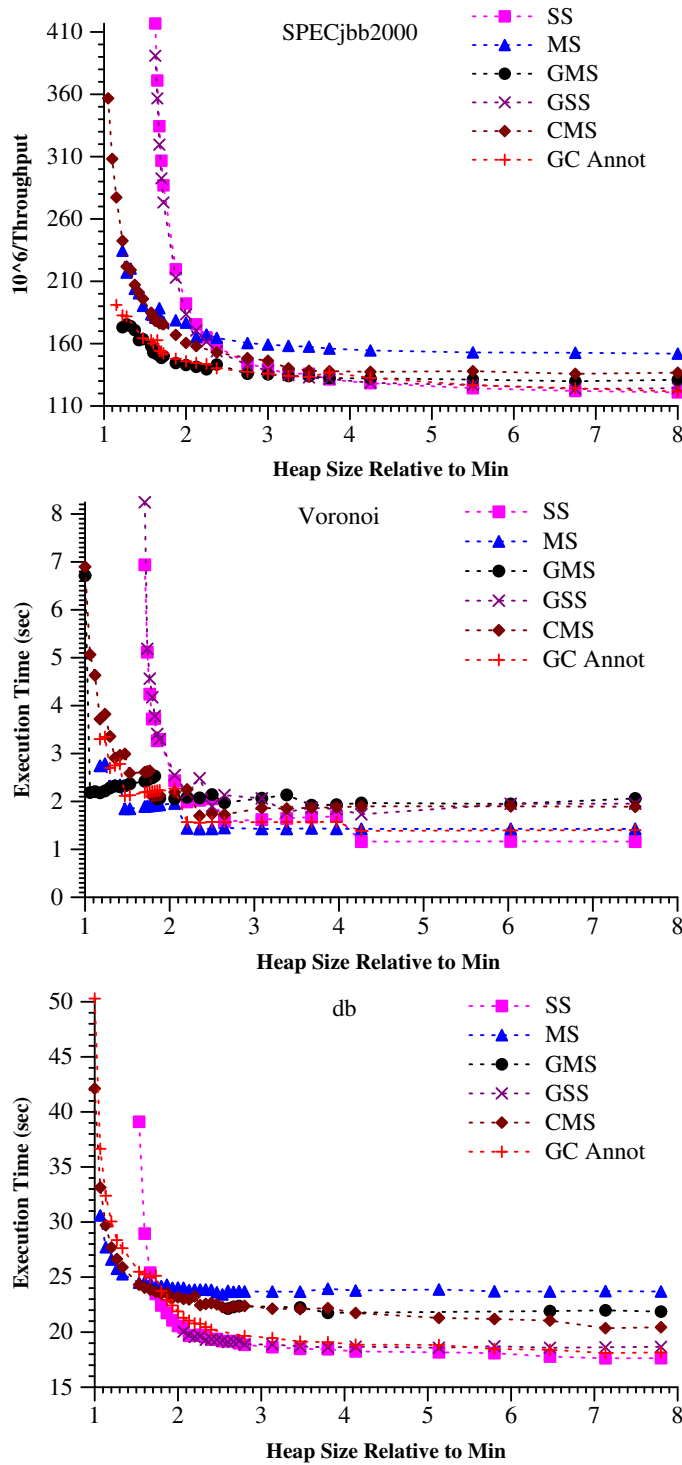


Fig. 13. Performance comparison between our switching system, GC Annot (dashed line with + marks), and the unmodified reference system built with five different GC systems. The figure shows three examples with switch points. The x-axis is heap size relative to the minimum (1 to 8 times the minimum). The y-axis is execution time (in seconds); for SPECjbb, the y-axis is the inverse of the throughput reported by the benchmark; we report $10^6/\text{throughput}$ in operations/second to maintain visual consistency with the execution time data. GC Annot effectively and efficiently tracks the best-performing collection system regardless of whether there are switch points.

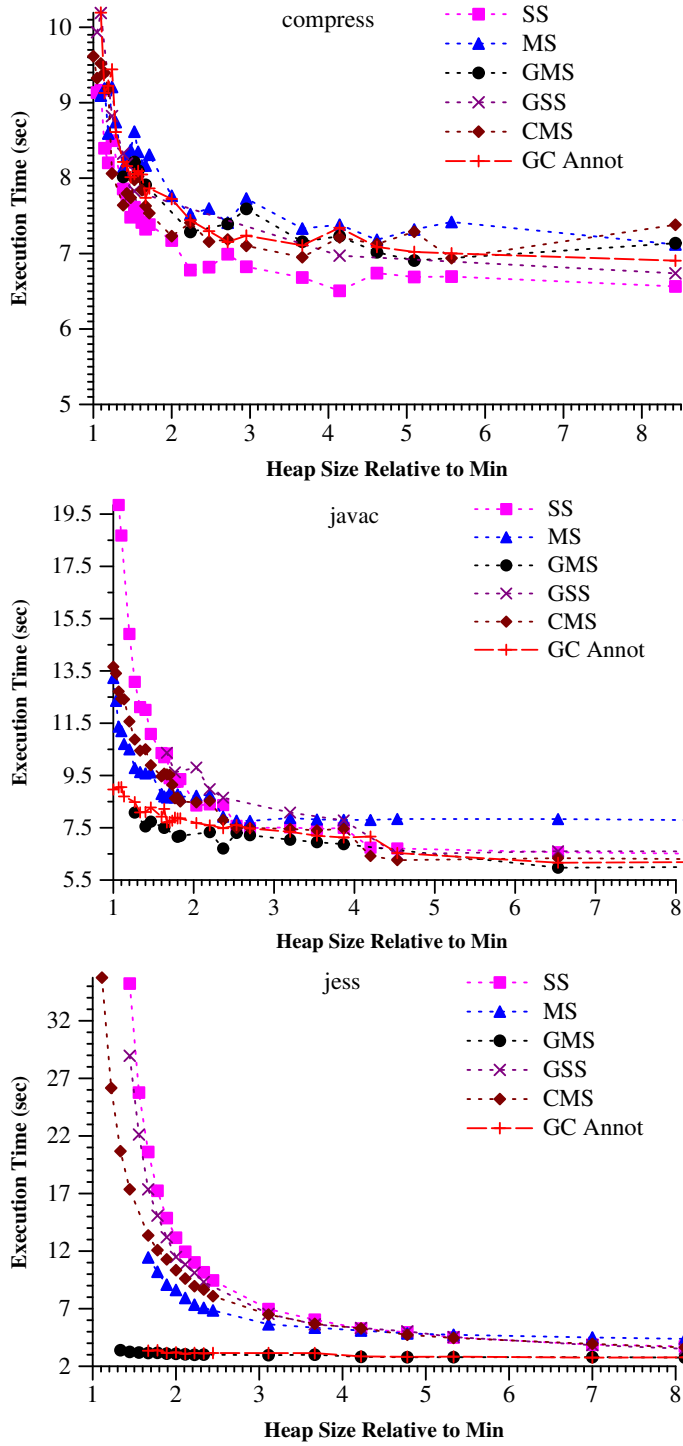


Fig. 14. Performance comparison between our switching system, GC Annot (dashed line with + marks), and the unmodified reference system built with five different GC systems. The figure shows three examples without switch points. The x-axis is heap size relative to the minimum (1 to 8 times the minimum). The y-axis is execution time (in seconds). GC Annot effectively and efficiently tracks the best-performing collection system regardless of whether there are switch points.

Average Difference Between Best & Worst GC Systems		
Benchmark	GCAnnot	
	Small Heaps (upto 3x)	
	Degradation over Best	Improvement over Worst
compress	6.65% (484ms)	2.85% (236ms)
jess	4.29% (132ms)	75.01% (10357ms)
db	3.54% (674ms)	8.48% (2108ms)
javac	6.55% (469ms)	27.55% (3626ms)
mtrt	1.31% (81ms)	47.02% (6024ms)
jack	3.34% (156ms)	40.92% (3722ms)
JavaGrande	4.77% (3088ms)	19.11% (17807ms)
SPECjbb	2.59% (3864*10 ⁶ /tput)	32.42% (106493*10 ⁶ /tput)
MST	3.83% (28ms)	56.80% (1244ms)
Voronoi	8.83% (164ms)	32.13% (1264ms)
Average	4.57%	34.23%

Fig. 15. Summarized performance differences between our annotation-guided switching system and the reference system for small heap sizes (minimum for an application to 3x the minimum). The table shows the percent degradation over the best- and percent improvement over the worst-performing GCs across small heap sizes (the time in milliseconds that this equates to is shown in parenthesis).

Average Difference Between Best & Worst GC Systems		
Benchmark	GCAnnot	
	Large Heaps (3x – 8x)	
	Degradation over Best	Improvement over Worst
compress	6.52% (432ms)	3.50% (258ms)
jess	2.04% (60ms)	44.11% (2378ms)
db	2.58% (469ms)	22.83% (5420ms)
javac	4.83% (314ms)	13.40% (1052ms)
mtrt	5.37% (320ms)	27.07% (2364ms)
jack	3.48% (152ms)	14.26% (756ms)
JavaGrande	3.68% (2275ms)	14.93% (11204ms)
SPECjbb	1.77% (2258*10 ⁶ /tput)	16.13% (24936*10 ⁶ /tput)
MST	4.38% (32ms)	27.38% (318ms)
Voronoi	7.87% (96ms)	30.09% (602ms)
Average	4.25%	21.37%

Fig. 16. Summarized performance differences between our annotation-guided switching system and the reference system for medium to large heap sizes (from 3x the minimum for an application to 8x the minimum). The table shows the percent degradation over the best- and percent improvement over the worst-performing GCs across medium to large heap sizes (the time in milliseconds that this equates to is shown in parenthesis).

Benchmark	GC Annot: Average Degradation Over Generational Mark-Sweep	
	compress	-0.37%
jess	2.82%	(85ms)
db	-14.17%	(-3122ms)
javac	5.19%	(373ms)
mtrt	2.32%	(78ms)
jack	3.22%	(147ms)
JavaGrande	-0.19%	(-87ms)
SPECjbb	0.95%	($1.72 \cdot 10^6$ / tput)
MST	-44.66%	(-827ms)
Voronoi	-11.88%	(-241ms)
Average	-5.68%	

Fig. 17. Percent degradation of our system over always using the widely used GMS collection. The negative values indicate that on average across heap sizes, our system improves performance over GMS.

over optimal GC selection.

Note that the data in these tables does not compare our system against a single Jikes RVM GC; instead, we are comparing our system against the *best-and-worst-performing GC 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 the 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.

We also collected the same results for when we omit Mark-Sweep (MS) collection. MS works well for small heaps but is thought to implement obsolete technology. On average across benchmarks and heap sizes, our system imposes 3% overhead over the best-performing GC at each point. In addition, our system reduces the overhead of selecting the worst-performing collector by 21 – 34% (depending on the heap size). Interestingly, when MS is not available in the system, the average degradation *decreases*. This is due to the fact that MS is the best-performing collector in a number of cases in which small and medium sized heaps are used.

Figure 17 presents the percent degradation over *always using the Generational/Mark-Sweep Hybrid (GMS)*. GMS is thought to be the best-performing, Jikes RVM GC – it is the default collector in Jikes RVM version that we extended. However, our data shows that it does not work well for all programs for all heap sizes. Our system enables a 6% improvement (a negative degradation) over always using GMS across benchmarks and heap sizes. This improvement

varies across inputs: 14% and 12% for db and Voronoi, to almost 45% for MST. Note, however, that MST is a very short running program – small differences in execution time (800ms) translate into very large percent differences. The improvement in db translates to a benefit of over 3 seconds.

Overall, these results indicate that our framework is able to achieve performance that is similar to the best-performing collector (in terms of both execution performance and compilation overhead) by making use of the annotations to guide dynamic switching between GCs. Moreover, when there is a switch point for programs, our system can enable the best performance on average over any single GC for that program. For cases in which there is no crossover between optimal collectors, our system maintains performance similar to that of the reference system. However, since the optimal GC varies across benchmarks, our system is able to perform better than any single GC across benchmarks.

Automatic Switching

We next evaluate the effectiveness of automatically switching between GCs using online program behavior and simple heuristics. Automatic switching requires the use of method invalidation and OSR to maintain correctness given the use of aggressive specializations: including/avoiding write barriers and inlining allocation routines – for the currently available, underlying GC. Our system employs our new version of OSR to enable both high performance and correctness.

The automatic switching scenario that we investigate addresses what happens when there is suddenly a loss of memory availability, i.e., the OS reclaims memory from the JVM for use by another, high-priority, application. In such a case, automatic switching can avoid OutOfMemory errors (or prevent excessive paging) by switching to a GC that works well when resources are constrained. We investigated the case in which memory was reduced to a point that the program can still make progress. For such cases, by switching to a more appropriate GC, we can reduce the overhead of garbage collection and improve performance.

We consider the situation in which after program startup, the OS reclaims memory such that the resulting heap size is twice the size of the reserved space (live data) following a garbage collection. We start with a maximum heap size of 200MB. We trigger heap resizing when the program steady state begins – which we approximate by 100 thread switches (we use 500 for specjbb since it is a longer running program). The switching system decides to switch when the GC load (defined in section 5) remains high for multiple GC cycles (we use three in the results). In addition, the system also switches when it observes that GCs are being triggered too frequently, measured as the duration

Benchmark	Base	Autoswitch	Pct. Impr.	# OSRs	OSR Time (ms)	Heapsize (MB)
compress	7.65	7.65	0.00	--	--	60
jess	7.23	3.89	46.20	10	28.46	28
db	31.29	23.16	25.98	1	1.39	24
javac	11.73	10.72	8.61	10	22.45	47
mtrt	24.77	9.11	63.22	2	58.35	24
jack	7.53	5.36	28.82	4	6.50	32
SPECjbb	175.10	158.70	9.71	0	0.00	100
JavaGrande	102.09	76.80	24.76	1	1.58	24
MST	0.94	0.94	0.00	--	--	100
Voronoi	4.37	3.94	9.15	2	3.00	60
Average	37.27	30.03	21.65	4	15.22	50
Average Spec98	15.03	9.98	28.81	5	23.43	36

Fig. 18. Performance of automatic switching when memory resources are suddenly constrained. Columns 2 and 3 show the time in seconds for execution for the clean (Base) system and our automatic switching system (including all overheads). Column 4 show the percent improvement enabled by our system. The right half of the table shows the OSR statistics: number of OSRs, total OSR time in milliseconds, and the heap size following the memory reclamation by the system.

for which application threads execute between successive garbage collections (we choose 300ms as the minimum application duration observed over 3 GC cycles).

We present the performance of this scenario in Figure 18. Columns 2 and 3 show the time in seconds for execution for the clean (Base) system and our automatic switching system (including all overheads). Column 4 shows the percent improvement enabled by our system. On average, our GC switching system can improve the performance of the program given dynamically changing resource conditions by over 29%. For the SpecJVM98 benchmarks, we improve performance by 22% on average. Interestingly, for some benchmarks, we found that Generational Mark-Sweep (GMS) incurs more garbage collections compared to always executing the application with Semispace. Yet, switching to GMS benefits the application since the total GC time is less compared to Semispace, since on average, a single GMS collection runs for a very short duration (as low as 9 milliseconds) compared to a typical Semispace collection (150 to 200 milliseconds). compress and MST do not allocate enough for a switch to be triggered. The right half of the table shows the OSR statistics. Column 5 is the number of OSRs, column 6 is the total OSR time in milliseconds, and column 7 is the heap size following the system memory reclamation.

In summary, automatic GC switching has the potential for enabling the application to make progress and avoid OutOfMemory errors if resources become constrained during program execution. In addition, it improves performance under such conditions by switching to a GC that imposes less GC overhead. Should memory availability be restored, our system can switch to a collector that performs well for large heap sizes, e.g., SS. Given the ability to dynamically and efficiently switch between competing collection systems, we now

have the ability to consider other mechanisms (e.g., program phase and data locality behavior) for deciding when to switch and to which GC we should switch to. We plan to investigate such techniques in future work.

7 Related Work

This paper is an extension of our initial work on application-specific garbage collection (Soman et al., 2004). The extensions to this prior work that we present herein include an extensive evaluation of the various components of our system as well as novel implementations of automatic switching and on-stack replacement (OSR).

On-stack replacement (OSR) was initially conceived of by the researchers and engineers of the Self-91 system (Chambers and Ungar, 1991; Ungar and Smith, 1987). The system employed OSR to defer compilation of uncommon code until its initial execution, to increase optimization opportunities, and to reduce compiled code space and compilation overhead. OSR has also been employed by modern dynamic virtual execution environments for dynamically de-optimizing code (for debugging purposes) (Hölzle et al., 1992), to defer compilation of method regions to avoid compilation overhead (and improve data flow) (Chambers and Ungar, 1991; Sukanuma et al., 2003; Fink and Qian, 2003; Sun Microsystems, Inc., 2001; Paleczny et al., 2001), and to optimize (promote) methods that executed unoptimized for a long time without returning (Hölzle and Ungar, 1994; Fink and Qian, 2003; Sun Microsystems, Inc., 2001; Paleczny et al., 2001) at thread switch points (commonly method prologues, epilogues, and loop back-edges). Jikes RVM implementation of OSR (Fink and Qian, 2003) that we extend herein (to be more amenable to compiler optimizations) is representative of prior approaches to OSR.

Two other areas of related work show that performance due to the GC employed varies across applications and that switching collectors dynamically can be effective. Prior work shows that performance can be improved by combining variants of the same collector in a single system (Lang and Dupont, 1987; Printezis, 2001), e.g., mark-and-sweep and mark-and-compact; and semispace and slide-compact. Sansom (Sansom, 1992) shows that coupling compaction with a semispace collector can be effective. No extant system, to our knowledge, provides a general, easily extensible framework that enables dynamic switching between a number of diverse collectors.

Other related work shows empirically that performance enabled by garbage collection is application-dependent. For example, Fitzgerald and Tarditi (Fitzgerald and Tarditi, 2000) performed a detailed study comparing the relative performance of applications using several variants of generational and non-generational copying collectors (the variations had to do with the write barrier implementations). They showed that over a collection of 20 bench-

marks, each collector variant sometimes provided the best performance. On the basis of these measurements they argued for profile-directed selection of GCs. However, they did not consider variations in input, required different pre-built binaries for each collector, and only examined semispace copying collectors.

Other studies have identified similar opportunities (Attanasio et al., 2001; Zorn, 1990; Smith and Morrisett, 1998). IBM’s Persistent Reusable JVM attempts to split the heap into multiple parts grouped by their expected lifetimes, employs heap-specific GC models and heap-expansion to avoid GCs. It supports command-line GC policies to allow the user to choose between optimizing throughput or average pause time. BEA’s Weblogic JRockit VM (BEA Systems Inc., 2003) employs an adaptive GC system which performs dynamic heap resizing. It also automatically chooses the collection policy to optimize for either minimum pause time or maximum throughput, choosing between concurrent and parallel GC, or generational and single-spaced GC, based on the application developer’s choice. BEA’s white-paper (BEA Systems Inc., 2003), however, describes the system at a very high level and provides few details or performance data. We were unable to compare our system against JRockit, due to its proprietary nature. To our knowledge, no extant research has defined and evaluated a general framework for switching between very diverse GCs, such as the one that we describe. In addition, our automatic switching scenarios are simple and require no user intervention to achieve considerable performance improvement.

Several prior approaches aim at finding the optimal heap size that would enables good performance, or mitigate the performance degradation caused by paging (Yang et al., 2004; Zhang et al., 2006). However, even if the virtual machine is able to select the optimal heap size given current resource restrictions, we believe that choosing the wrong collector, can yet hurt performance.

Sachindran et al present the MC^2 collector (Sachindran et al., 2004), which enables better performance compared to existing JikesRVM GCs. At the time of this writing, this collector was not yet freely available for our study. Our focus in this work is not to develop a new garbage collection system. Given existing, commonly and freely available GCs, we discuss the issues involved in efficiently switching between these. Even if in the future, a single collector does achieve best performance across applications, it is unclear whether it will do so in a variety of different application domains, environments, and for *every single* application. We believe that future systems that are designed to execute a variety of applications, would require the functionality to switch between collectors, even if there were a single collector that achieves good performance in *most* cases.

8 Conclusions and Future Work

Garbage collection plays an increasingly important role in next generation Internet computing and server software technologies. However, the performance of collection systems is largely dependent upon application execution behavior and resource availability. In addition, the overhead introduced by selection of the “wrong” GC can be significant. To overcome these limitations, we have developed a framework that can automatically switch between GCs 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.

We present specialization techniques that enable the system to be very low overhead and to achieve significant performance improvements over traditional, non-switching, virtual execution environments. We describe a novel implementation of on-stack replacement (OSR) that can enable efficient replacement of executing code at any point in the program at which a GC (and thus a GC switch) can occur. Our results show that, on average, our OSR implementation reduces the overhead of our system by 9% over an extant approach to OSR (Figure 10(a)).

We also present two techniques that exploit the efficient GC switching functionality. In particular, we describe and present the effectiveness of annotation-guided (based on offline profiling) and automatic (based on online profiling) switching. We empirically evaluate our system using a wide range of heap sizes, benchmarks, and scenarios. Our results show that, on average, annotation-guided GC switching introduces around 4% overhead and improves performance by 21 – 34% over the worst-performing GC (depending on the heap size). Moreover, our automatic switching system can avoid OutOfMemory errors and improve performance by 22% on average for a scenario in which memory resources are dynamically reclaimed by the OS for another process while the program is executing.

As part of future work, we plan to investigate techniques that reduce the overhead of automatic switching, and dynamically identify switch points online. We plan to consider the frequency of collections, allocation rates, and memory hierarchy behavior to guide adaptive selection of collection and allocation algorithms. In addition, we plan to investigate how our system adapts to variable workloads in a server environment that runs multiple applications simultaneously.

References

AIKEN, A. AND GAY, D. 1998. Memory Management with Explicit Regions.

- In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 313–323.
- ALPERN, B., ATTANASIO, C., BARTON, J., BURKE, M., CHENG, P., CHOI, J.-D., COCCHI, A., FINK, S., GROVE, D., HIND, M., HUMMEL, S., LIEBER, D., LITVINOV, V., MERGEN, M., NGO, T., RUSSELL, J., SARKAR, V., SERRANO, M., SHEPHERD, J., SMITH, S., SREEDHAR, V., SRINIVASAN, H., AND WHALEY, J. 2000. The Jalapeño Virtual Machine. *IBM Systems Journal* 39, 1, 211–221.
- APPEL, A. W. 1989. Simple Generational Garbage Collection and Fast Allocation. *Software Practice and Experience* 19, 2, 171–183.
- ATTANASIO, C., BACON, D., COCCHI, A., AND SMITH, S. 2001. A Comparative Evaluation of Parallel Garbage Collectors. In *Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 2624. Springer-Verlag, 177–192.
- BACON, D., ATTANASIO, C., H., RAJAN, V., AND SMITH, S. 2001. Java without the Coffee Breaks: A Non-intrusive Multiprocessor Garbage Collector. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 92–103.
- BACON, D., FINK, S., AND GROVE, D. 2002. Space- and Time-Efficient Implementation of the Java Object Model. In *European Conference on Object-Oriented Programming (ECOOP)*. 111–132.
- BEA SYSTEMS INC. 2002. BEA’s Enterprise Platform. IDC white paper by M. Rosen sponsored by BEA. http://ww.bea.com/content/news_events/white_papers/BEA_Beyond_Applicati%on_Server_wp.pdf.
- BEA SYSTEMS INC. 2003. BEA Weblogic JRockit: Java For the Enterprise. http://www.bea.com/content/news_events/white_papers/BEA_JRockit_wp.pdf.
- BLACKBURN, S., CHENG, P., AND MCKINLEY, K. 2003. A Garbage Collection Design and Bakeoff in JMTk: An Efficient Extensible Java Memory Management Toolkit. Tech. Rep. TR-CS-03-02, Department of Computer Science, FEIT, ANU. Feb. <http://eprints.anu.edu.au/archive/00001986/>.
- BLACKBURN, S., SINGHAI, S., HERTZ, M., MCKINLEY, K., AND MOSS, J. 2001. Pretenuing for Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 342–352.
- BLACKBURN, S. M., JONES, R., MCKINLEY, K. S., AND MOSS, J. E. B. 2002. Beltway: Getting Around Garbage Collection Gridlock. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 153–164.
- BRECHT, T., ARJOMANDI, E., LI, C., AND PHAM, H. 2001. Controlling Garbage Collection and Heap Growth to Reduce Execution Time of Java Applications. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 353–366.
- CAHOON, B. AND MCKINLEY, K. 2001. Data Flow Analysis for Software Prefetching Linked Data Structures in Java Controller. In *International*

- Conference on Parallel Architectures and Compilation Techniques*. 280–291.
- CHAMBERS, C. AND UNGAR, D. 1991. Making Pure Object-Oriented Languages Practical. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Vol. 26. ACM Press, 1–15.
- CIERNIAK, M., LUEH, G., AND STICHNOTH, J. 2000. Practicing JUDO: Java Under Dynamic Optimizations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 13–26.
- CLINGER, W. AND HANSEN, L. T. 1997. Generational Garbage Collection and the Radioactive Decay Model. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 97–108.
- DIWAN, A., MOSS, J., AND HUDSON, R. 1992. Compiler Support for Garbage Collection in a Statically Typed Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
- FINK, S. AND QIAN, F. 2003. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *International Symposium on Code Generation and Optimization (CGO)*. 241–252.
- FITZGERALD, R. AND TARDITI, D. 2000. The Case for Profile-Directed Selection of Garbage Collectors. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)*. 111–120.
- HALLOGRAM JRUN. 2001. JRun Execution Environment. Project home page. <http://www.hallogram.com/jrun>.
- HERTZ, M., FENG, Y., AND BERGER, E. D. 2005. Garbage Collection Without Paging. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
- HEWLETT-PACKARD NONSTOP. 2003. NonStop Server for Java Software from Hewlett-Packard Corporation. Project home page. <http://h20223.www2.hp.com/NonStopComputing/cache/76698-0-0-0-121.html>.
- HICKS, M., HORNOF, L., MOORE, J., AND NETTLES, S. 1998. A Study of Large Object Spaces. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)*. 138–145.
- HIRZEL, M., DIWAN, A., AND HERTZ, M. 2003. Connectivity-based Garbage Collection. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 359–373.
- HÖLZLE, U. 1994. Optimizing Dynamically Dispatched Calls with Run-Time Type Feedback. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 326–336.
- HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 32–43.
- HÖLZLE, U. AND UNGAR, D. 1994. A Third Generation Self Implementation: Reconciling Responsiveness With Performance. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 229–243.
- IBM SHIRAZ. 2001. IBM’s Persistent Reusable JVM. Project

- home page. <http://www.haifa.il.ibm.com/projects/systems/rs/persistent.html>.
- IBM WEBSHERE. 2004. The WebSphere Software Platform. Product home page. <http://www-3.ibm.com/software/info1/websphere/index.jsp>.
- JAVA GRANDE FORUM. 1998. The Java Grande Forum. <http://www.javagrande.org/>.
- JONES, R. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley.
- KRINTZ, C. 2003. Coupling On-Line and Off-Line Profile Information to Improve Program Performance. In *International Symposium on Code Generation and Optimization (CGO)*. 69–78.
- KRINTZ, C. AND CALDER, B. 2001. Using Annotation to Reduce Dynamic Optimization Time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 156–167.
- LANG, B. AND DUPONT, F. 1987. Incrementally Compacting Garbage Collection. In *Symposium on Interpreters and Interpretive Techniques*. 253–263.
- LINDHOLM, T. AND YELLIN, F. 1999. *The Java Virtual Machine Specification*, Second ed. Addison Wesley.
- PALECZNY, M., VICK, C., AND CLICK, C. 2001. The Java HotSpot(TM) Server Compiler. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*. 1–12.
- PRINTEZIS, T. 2001. 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*. 171–184.
- SACHINDRAN, N. AND MOSS, J. E. B. 2003. Mark-copy: Fast Copying GC With Less Space Overhead. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 326–343.
- SACHINDRAN, N., MOSS, J. E. B., AND BERGER, E. D. 2004. MC^2 : High-performance Garbage Collection for Memory-constrained Environments. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.
- SANSOM, P. 1992. Combining Single-Space and Two-Space Compacting Garbage Collectors. In *Glasgow Workshop on Functional Programming*. 312–323.
- SMITH, F. AND MORRISETT, G. 1998. Comparing Mostly-Copying and Mark-Sweep Conservative Collection. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)*. 68–78.
- SOMAN, S., KRINTZ, C., AND BACON, D. F. 2004. Dynamic Selection of Application-Specific Garbage Collectors. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)*. 49–60.
- SPEC Corporation 1998. Standard Performance Evaluation Corporation (SpecJVM98 and SpecJBB Benchmarks). <http://www.spec.org/>.
- SUGANUMA, T., YASUE, T., AND NAKATANI, T. 2003. A Region-Based Compilation Technique for a Java Just-In-Time Compiler. In *ACM SIGPLAN*

- Conference on Programming Language Design and Implementation (PLDI)*. 312–323.
- SUN MICROSYSTEMS, INC. 2001. The Java HotSpot Virtual Machine, Technical White Paper. http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_%4_30_01.ps.
- UNGAR, D. 1992. Generation Scavenging: A Non-Disruptive High Performance Storage Recalamation Algorithm. In *Software Engineering Symposium on Practical Software Development Environments*. 157–167.
- UNGAR, D. AND JACKSON, F. 1992. An Adaptive Tenuring Policy for Generation Scavengers. *ACM Transactions on Programming Languages and Systems* 14, 1, 1–27.
- UNGAR, D. AND SMITH, R. 1987. Self: The Power of Simplicity. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 227–242.
- YANG, T., HERTZ, M., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. 2004. Automatic Heap Sizing: Taking Real Memory Into Account. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)*.
- ZHANG, C., KELSEY, K., SHEN, X., DING, C., HERTZ, M., AND OGIHARA, M. 2006. Program-level Adaptive Memory Management. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)*.
- ZORN, B. 1990. Comparing Mark-And-Sweep and Stop-And-Copy Garbage Collection. In *ACM Conference on LISP and Functional Programming*. 87–98.