Dynamic Selection of Application-Specific Garbage Collectors

Sunil Soman Chandra Krintz
Computer Science Department
University of California, Santa Barbara
Santa Barbara, CA 93106
{sunils,ckrintz}@cs.ucsb.edu

David F. Bacon
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
dfb@watson.ibm.com

ABSTRACT

Much prior work has shown that the performance enabled by garbage collection (GC) systems is highly dependent upon the behavior of the application as well as on the available resources. That is, no single GC enables the best performance for all programs and all heap sizes. To address this limitation, we present the design, implementation, and empirical evaluation of a novel Java Virtual Machine (JVM) extension that facilitates dynamic switching between a number of very different and popular garbage collectors. We also show how to exploit this functionality using annotation-guided GC selection and evaluate the system using a large number of benchmarks. In addition, we implement and evaluate a simple heuristic to investigate the efficacy of switching automatically. Our results show that, on average, our annotation-guided system introduces less than 4% overhead and improves performance by 24% over the worst-performing GC (across heap sizes) and by 7% over always using the popular Generational/Mark-Sweep hybrid.

Categories and Subject Descriptors

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

General Terms

Design, Performance, Experimentation, Algorithms

Keywords

Application-specific collection, dynamic selection, hot-swapping, annotation, virtual machine, Java

1. INTRODUCTION

Garbage collection is a mechanism for automatic reclamation of dynamically allocated memory. It simplifies the program development cycle by eliminating the burden of explicit memory deallocation. However, garbage collection imposes a performance overhead since it must identify and reuse memory that is no longer accessible by the program, *while* the program is executing.

ISMM'04, October 24–25, 2004, Vancouver, British Columbia, Canada.

The performance of heap allocation and collection techniques has been the focus of much recent research [11, 12, 10, 18, 1, 14, 41, 5, 19]. The goal of most of this prior work has been to provide general-purpose mechanisms that enable high-performance execution across all applications. However, other prior research [4, 16, 42, 36] has shown that the efficacy of a memory management system (the allocator and the garbage collector) is dependent upon application behavior and available resources. That is, no single collection system enables the best performance for all applications and all heap sizes. Our empirical experimentation confirms these findings. Over a wide-range of heap sizes and the 11 benchmarks studied, we found that each of *five different* collectors enabled the best performance at least once; this set of garbage collection systems includes those that implement semispace copying, generational collection, mark-sweep collection, and hybrids of these different systems. As such, we believe that to achieve the best performance, the collection and allocation algorithms used should be specific to both application behavior and available resources.

Existing execution environments enable application- and heap-specific garbage collection, through the use of different configurations (via separate builds or command-line options) of the execution environment. However, such systems do not lend themselves well to next-generation, high-performance server systems in which a single execution environment executes continuously while multiple applications and code components are uploaded by users [22, 17, 33]. For these systems, a single collector and allocator must be used for a wide range of available heap sizes and applications, e.g., e-commerce, agent-based, distributed, collaborative, etc. As such, it may not be possible to achieve high-performance in all cases and selection of the *wrong* GC system may result in significant performance degradation.

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

To exploit this dynamic switching functionality, we also present *annotation-guided GC selection* to enable application-specific garbage collection. With each Java program, we annotate the best-performing GC for a range of heap sizes. Upon invocation, the JVM switches to the GC system specified by the annotation given the current maximum available heap size. We identify the best-performing

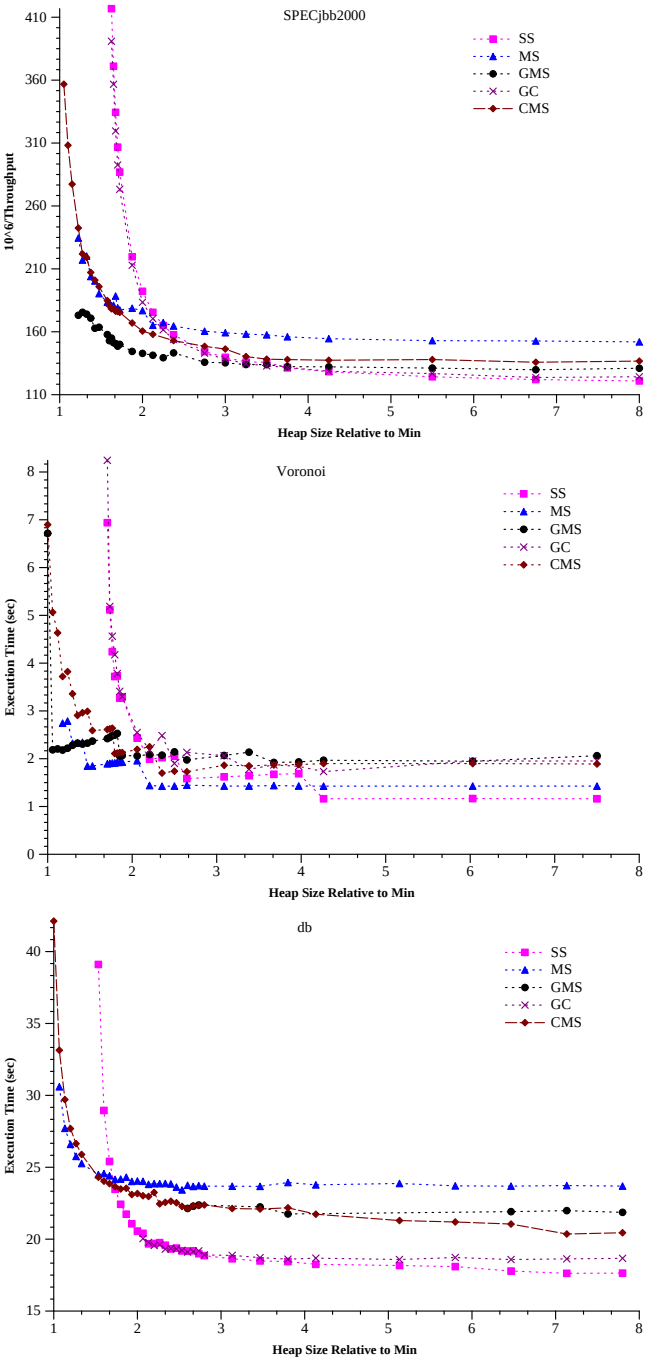

Figure 1: Benchmark performance using different GC systems and heap sizes in the JikesRVM. No single collector enables the best performance for all programs and all heap sizes. The GC systems that we used for these experiments are Semispace copying (SS), Mark-sweep (MS), Generational/Mark-sweep (GMS) hybrid, Generational Semispace (GSS), and a non-generational Semispace/Mark-sweep collector (CMS) (see Section 2.1 for details on these collectors). The y-axis is total time in seconds. For SPECjbb, the y-axis is the inverse of the throughput reported by the benchmark; we report 10^6/throughput in operations/second to maintain visual consistency with the other graphs. The x-axis is heap size relative to the minimum in which the program can execute. The benchmarks are from the SPECjbb, JOlden, and SpecJVM98 suites.

GC system, off-line, via execution profiles from multiple inputs and resource constraints. We then annotate the program with this information using a bytecode annotation system that we developed in prior work [28]. Furthermore, we extend this technique using a simple heuristic to investigate the efficacy of switching *automatically* using only on-line application and resource behavior.

To evaluate our system, we employed a number of different benchmarks. We present results from this evaluation and compare the overhead of our system (without switching) with that of a clean, unmodified system for a wide range of garbage collection configurations. In addition, we present the efficacy of annotation-guided GC selection and empirically evaluate automatic switching. Our results indicate that annotation-guided GC selection imposes very little overhead (3-4%) over using the best-performing GC system at each heap size. In addition, our system can significantly reduce the negative impact of selecting the *wrong* collector (24-26%) and improve performance over always using the popular Generational/Mark-Sweep hybrid (by 7% on average).

We next motivate our work and overview the design and implementation of our dynamic GC switching framework. In Section 3 we present annotation-guided and automatic switching strategies that exploit dynamic switching to improve program performance. We then describe and analyze our experimental results in Section 4, and present our related work (Section 5), and conclusion (Section 6).

2. APPLICATION-SPECIFIC GC

The next-generation of high-performance server systems must enable continuous availability and high-performance to gain widespread use and acceptance. Due to the portability, flexibility, and security features enabled by the Java programming language and its execution environments, a number of high-end server systems now employ Java as the implementation language for application and execution servers [22, 17, 33]. These systems run a single virtual machine (VM) image continuously so that applications and code components can be uploaded and executed as needed by customers (for customization, collaboration, distributed execution, etc.).

Given this model (single VM and continuous execution) and existing JVM technology, a single, general-purpose collector and allocation policy must be used for all applications. However, many researchers have shown that there is no single combination of a collector and an allocator that enables the best performance for all applications, on all hardware, and given all resource constraints [4, 16, 42]. Figure 1 confirms these findings. The graphs show performance over heap size for SPECjbb [38], Voronoi from the JOlden benchmark suite [13], and db from the SpecJVM98 suite [38] executing within the Jikes Research Virtual Machine (JikesRVM) [2]. The x-axis represents heap size relative to the minimum heap size that the application requires for complete execution. For SPECjbb, the y-axis is the inverse of the throughput reported by the benchmark; we report 10^6/throughput in operations per second to maintain visual consistency with the execution time data of the other benchmark. For the data in all graphs, lower values are better.

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 and CMS (a non-genera-

tional semispace copying/mark-sweep hybrid) and MS perform best for small heap sizes. We refer to any point at which the best-performing GC system changes as a *switch point*. These results support the findings of others [4, 16, 42], that no single collection system enables the best performance across benchmarks; moreover no single system performs best *across heap sizes for a single benchmark/input pair*.

To exploit this execution behavior that is specific to both the application and the underlying resource availability, we extended JikesRVM, to enable dynamic switching between GC systems. 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.

2.1 Implementation Framework

The JikesRVM [2] is an open-source, dynamic and adaptive optimization system for Java that was designed and continues to evolve with the goal of enabling high-performance in server systems. The JikesRVM compiles (at runtime) Java bytecode programs at the method-level, Just-In-Time, to x86 (or Power PC) code. The system performs extensive runtime services, e.g., garbage collection, thread scheduling, synchronization, etc. In addition, JikesRVM implements adaptive optimization by performing on-line instrumentation and profile collection and then using the profile data to evaluate when program characteristics have changed enough to warrant method-level re-optimization. The current version of the JikesRVM optimizing compiler applies two levels of optimization (0 and 1). 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 system 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).

The JikesRVM (version 2.2.0+) implements the Java Memory Management Toolkit (JMTk) [8] that enables garbage collection and allocation algorithms to be written and "plugged" into JikesRVM. The framework offers a high-level, uniform interface to JikesRVM that is implemented by all memory management routines. We refer to the combination of an allocation policy and a collection technique as a *GC system* (this corresponds to a *Plan* in JMTk terminology). The JMTk allows users to implement their own GC systems easily within JikesRVM and to perform an empirical comparison with other existing collectors and allocators. The JMTk provides users with utility routines for common GC operations, such as, copying, marking and sweeping objects. When a user builds a configuration of JikesRVM, she is able to select a particular GC system for incorporation into the JikesRVM image.

The five GC systems 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 pause when garbage collection is in progress. Readers should refer to [26, 8] for a detailed description of these collectors; however, the generational collectors deserve special mention.

The GSS system makes use of well-known generational garbage collection techniques [3, 40]. Young objects are allocated in a variable-sized nursery space using *bump-pointer* allocation from a contiguous block of memory. Upon a minor collection, the nursery is collected and the survivors are copied to the mature space. The

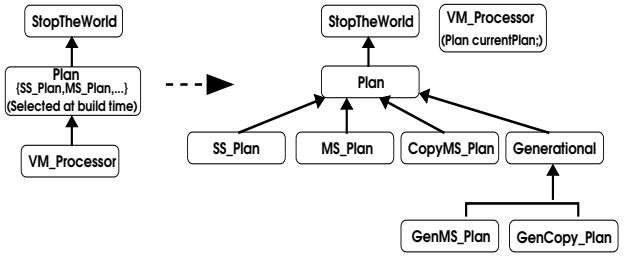

Figure 2: Original and new (dynamic, switch-enabled) JikesRVM/JMTk class hierarchy

mature space is collected by performing a semispace copying collection following a minor collection, as needed. This process is referred to as a major collection. Since generational GC performs minor collections separately from major collections, write-barriers are needed for all pointer stores to identify pointers from objects in the mature space to those in the nursery. We describe the implications of the use of write barriers in our system in Section 2.2.3.

The GMS system also employs a generational model. However, the mature space is managed in the same way as the mark-sweep space in the non-generational Mark-sweep system (MS). Allocation from this space is performed using a sequential, first-fit, free-list resource and 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.

CMS [1] is similar to SS in that it is non-generational and is divided into two sections. 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.

JVM system classes and large objects are handled specially in the JikesRVM/JMTk system. All of the GC systems include a immortal space that holds the JikesRVM system classes. Immortal space is allocated using the bump-pointer technique and this space is never collected. In addition, each collection system considers objects of size 16KB and greater as "large objects". Generational collectors allocate large objects from the mature space; non-generational collectors employ a separate large object space. This large object space, when used, is managed via Mark-sweep collection.

We extended the JikesRVM/JMTk system to implement all of these GC systems within a single image of the execution environment. Moreover, we enabled the system to switch between SS, GMS, GSS, CMS, and MS dynamically. In the following section, we describe the implementation of our GC switching system.

2.2 Multiple Garbage Collection Systems in a Single JVM

The version of the *Plan* class dictates which GC system is built into a single JikesRVM image. The only way to change *Plan*s (to use a different garbage collector) is to build another image using a different JikesRVM configuration. Our extension to JikesRVM requires that multiple GC systems be included in a single system image. To enable this, we implemented a generic *Plan* class, from

[1] Note: This is a stop-the-world collector and should not be confused with the Concurrent Mark-Sweep collector in Sun's HotSpot VM [23]

which all specific GC system classes derive, e.g., SSPlan, CMSPlan, GMSPlan, GSSPlan, etc. Each of these plans are instantiated in a single image of our system. Since the JMTk provides common utility routines, we can share most of the garbage collection code among the plans. The size of a typical VM image built with our extensions is 44.2MB, compared to an average size of 42.6MB for the reference JikesRVM images (ranging from 37.2MB for SS to 49.4MB for MS) – our extensions do not significantly increase code size. Figure 2 shows the JikesRVM JMTk class hierarchy before and after our extensions.

We inserted a global field (*currentPlan*), into the class that implements the JikesRVM GC system interface. This field identifies the GC system that is currently in use. At all times, the instantiation of each GC system (a *Plan* object) is available in our system.

Each program allocation site invokes the *alloc* method in the *Plan* class. We implemented this routine as a static method to avoid virtual method dispatch and guarded inlining. This method invokes the appropriate static allocation routine according to the GC system identified by *currentPlan*. When a switch occurs, we update *currentPlan* to reflect the GC system to which we have switched.

To support multiple GC systems, we require address ranges for all possible virtual memory resources to be reserved. Our goal is to enable as much overlap of virtual address spaces as possible to reduce the overhead of switching (described further below). The address space layout that we use is shown in Figure 3. Each address range is mapped to physical memory lazily (as it is used by the executing program), in 1 Megabyte chunks. The immortal and large object space within our system are shared across all GC systems. Similar to the reference system, we allocate objects larger than 16KB from the large object space.

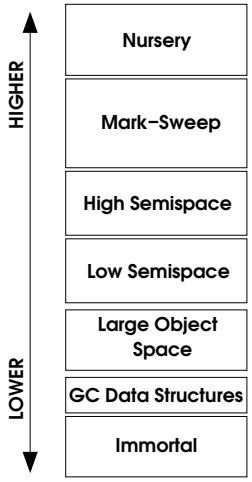

Figure 3: Virtual address layout in the switching system

2.2.1 Switching Between GC Systems

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

When a switch occurs, we stop each executing mutator thread as if a garbage collection was taking place. A full garbage collection, however, may not be necessary for all switches. To enable this, we meticulously designed the layout of our heap spaces (Figure 3) in such a way so 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 come from 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.

Table 1 indicates when a GC is required on a switch and if it is, what type of GC is required, e.g., full (F) or minor (M). If no GC is required then the entry in the matrix is marked with an N. We use the notation XX–>YY to indicate a switch from collection system XX to collection system YY; in the matrix, the entries show the type of GC that is required for row–>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). 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 since only switches to SS and GSS will require a full garbage collection.

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

Switches That Do Not Require Collection. As mentioned above, SS–>GSS does not require a collection since their virtual semispaces are shared. Similarly, MS–>GMS, MS–>CMS, and GMS–>CMS do not require a garbage collection upon a switch since the mark-sweep is shared. Following the update to *currentPlan*, our system allocates heap space from the nursery or initial semispace, for GSS and GMS, and from the initial semispace for CMS.

Switches That Require Minor Collection. When we switch from a generational to a similar non-generational collector, e.g., GMS–>MS and GSS–>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–>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.
- **GMS/MS–>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 2.2.2.
- **CMS–>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 system, we must perform a full collection to ensure that we consider all live objects.

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

GC Requirements Upon Switch N:None, F:Full, M:Minor Switch from Row to Column					
	SS	CMS	GMS	GSS	MS
SS	N	F	F	N	F
CMS	F	N	F	F	F
GMS	F	N	N	F	M
GSS	M	F	F	N	F
MS	F	N	N	F	N

Table 1: Given the layout of our virtual address spaces, we may or may not need to perform a full garbage collection for all switches. The entries in this table indicate when GC is required on a switch (from the row GC to the column GC) and if it is, what type of GC is required: full (F), minor (M), or none (N).

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. The reference JikesRVM implementation uses on-demand memory mapping of the virtual address space. To use physical memory efficiently, we dynamically unmap unused memory space when we switch to a new collection system.

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 physical memory than the reference system, *while* performing the switch in these cases. In practice however, our system never performs a switch from MS to SS or GSS when memory is constrained (we provide further explanation of why this is the case in Section 3). 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.

2.2.2 Multi-purpose Object Header

As mentioned in the previous section, to switch from a GC system that uses a mark-sweep space (GMS, CMS, and MS) to a GC system that uses a contiguous semispace (GSS, SS), we must maintain state for *both* the mark-sweep process as well as for the process of forwarding objects to the semispace. Typically, garbage collectors store this state in the header of each object. In JikesRVM, 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* by this GC system 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.

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

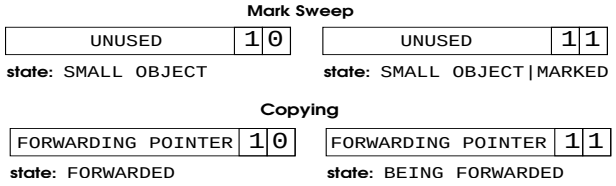

Figure 4: Examples of bit positions in status word in object header

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 with which JikesRVM is configured at build-time. For example, as shown in Figure 4, if JikesRVM is built using a mark-sweep GC system, 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 these bits are both 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 concurrently, 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 [6]) in which we "steal" the two least-significant bits from another address value that is byte-aligned. In JikesRVM, the object header also stores a pointer to a Type Information Block (TIB), 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 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 disregarded.

2.2.3 Optimizations

There are two primary sources of overhead introduced by our automatic GC switching system: The use of write barriers which are not needed by all collectors and the loss of inlining opportunities due to dynamically changing allocation routines. As mentioned in Section 2.2.3, since our system can switch to a generational collector at anytime, we must insert write-barriers for every field assignment in every method – these instructions execute even when the collector in use is non-generational. Moreover, if there is ever only a single GC system, we can inline calls to the allocation routine. However, in our system the allocation routine may change; consequently, we cannot perform such inlining.

To address these issues, we implemented two compiler optimizations that enable *specialization* of optimized methods according to the underlying GC system. In particular, for methods optimized at level 1 (as is done in the base system), we inline the allocation routines of current GC system. In addition, for methods optimized at level 0 or 1 (as is done in the base system), we only insert write barriers for field accesses in methods when the current GC is generational.

As mentioned previously, we employ adaptive optimization in which only hot methods are optimized at increasing optimization levels (only 0 and 1 currently). Hot methods are identified via online

sampling. All other methods are fast-compiled. We modified the fast compiler to insert write barriers into all methods regardless of the underlying GC system. In addition, the fast compiler performs no inlining.

If our system switches to a new GC system *after* method specialization has occurred, we must invalidate these methods so that future invocation of them will cause recompilation. We implement invalidation by simply replacing the table entry for the method with the same compilation stub used by JikesRVM to enable lazy, method-level, compilation [29]. Moreover, if such a method is currently executing, i.e., it is on the runtime stack of some application thread, we must recompile it immediately (as part of the switch) and replace its runtime stack frame.

To enable this, we implemented a general form of the on-stack replacement and method invalidation mechanism described in [15]. A complete description of our implementation is described in the technical report version of this paper [37]. In summary, we extended the system to enable OSR at any point during program execution – at which a GC system switch can occur, i.e., program points at which the compiler has generated live-variable information. These points include allocation sites, call sites, prologue and epilogue yieldpoints, loop backedges, and explicit yieldpoints [2].

We employ *OSRInfo* points to gather the information consisting of the stack and method-local variables, required to perform OSR, at every potential switch point. *OSRInfo* points are similar to *OSRPoints* used in [15] in that they *OSRInfo* points record information about live variables. However, they are not *unconditional yield points*, unlike *OSRPoints*. Moreover, *OSRInfo* points are only used during compilation to generate liveness information about variables at all garbage collection switch points, and as such, do not appear in the final machine code. During on-stack replacement, the switching system accesses the maps generated from the OSR information to extract values from the correct register and stack locations for the method being replaced. The OSR system then inserts these values into the appropriate frame locations for the new version of the method.

In addition to method invalidation and OSR, we implemented the write barrier to be as efficient as possible. We use a single, shared write-barrier for all types of generational GC systems. Moreover, as we showed in Figure 3, we placed the nursery space in our system at the highest virtual address. Hence, we require only a single check (to determine if the value is greater than the nursery boundary) to determine if the young object reference is in the nursery. Our universal write-barrier implementation is similar to that described in [9].

3. GARBAGE COLLECTOR SELECTION

By implementing the functionality to switch between collection systems while the JikesRVM is executing, we can now select the "best-performing" collection system for each application that executes using our system. To this end, we implemented *Annotation-guided GC System Selection*. Annotation is information that is communicated (transfered and loaded) as part of the program; this information can include anything from static analysis data to offline profile information. Annotation has be shown to be effective for reducing the overhead of dynamic optimization and for improving execution performance by providing "hints" to the compilation and runtime environment about optimization opportunities and analysis information [27, 28, 31, 20, 25]. We use annotation in this work to identify per-application garbage collectors that should be employed by our GC Switching system.

To enable annotation-guided GC selection, we analyzed application performance off-line using the different JikesRVM GC systems. We considered a number of different heap sizes and program inputs. We list these inputs in our annotation selection table Table 2, as input and cross. We extracted, for each heap size, the best performing GC system 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 (if any), i.e., the heap sizes at which the best-performing GC changes.

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	—
mpegaudio	100/10	11	GSS	—
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
specjbb	1 warehouse/2	40	GMS/SS	3.00
voronoi	65000 pts/20000	34	MS/SS	4.26

Table 2: Annotated GC selection decisions, minimum heap sizes and inputs. If there is a switch point, we annotate the minimum heap size and the *Switch Point Ratio*: switch point heap size over minimum heap size. We present data for the first input (Input) in the pair Input/Cross. We consider both inputs (Input & Cross) to infer the best performing GC at a particular heap size/minimum heap size ratio.

For all of the benchmarks that we studied, the per-GC performance was very similar across inputs. Only two benchmarks exhibited differences in GC performance across inputs. This *input-dependence* is very different from other types of profiles, e.g., method invocation counts, field accesses, etc., in which cross-input behavior can vary widely [27]. Therefore, it is less likely that we will negatively impact performance for inputs that we have not profiled. Even so, we consider two inputs for each benchmark to determine the annotation. To compute the GC to annotate for the two benchmarks that exhibited differences across inputs, we identified the GC that imposed the smallest percent degradation over the best performing collector across inputs at each heap size.

The values that we annotate are shown in the final two columns of Table 2. For each benchmark, we specify the GC system 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 switch points.

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 of current maximum heap size to minimum heap size and compares this value with the annotated ratio. If the value is less, the JVM switches to the first GC (left of slash in the table entry) and to the second GC (right of slash), otherwise. This requires that we also annotate the minimum heap size for the program and input. However, this reduces the amount of offline profiling required since given the minimum heap size for an input, we can compute the switch point using the ratio from any input, since this switch point ratio holds across inputs for all benchmarks that we studied. Five of these eleven programs have switch points.

We use a 4-byte annotation in each class file of an application containing a main method. We insert annotations into class files using an annotation language and a highly compact encoding that we

developed in prior work [28]. Upon initiation of dynamic loading of the first application class file, JikesRVM switches to the collection system specified; if there is a switch point for the program, our system compares the switch point ratio with the ratio of current maximum available heap size and minimum heap size. If the minimum heap size is not specified, 40MB is assumed. Since the best-performing collection system may depend on the underlying architecture (memory size, cache levels, cache sizes, register count), we can also incorporate different architectures as part of our profile collection and annotation. For this work, we focus solely on the x86 architecture.

One limitation of this annotation-based approach, is input dependence. Even though we consider multiple inputs and for these benchmarks, the gc-selection variance across inputs is small, it may happen that a previously unknown input causes the selection decision to change – needlessly causing a degradation in program performance. Moreover, offline profiling requires more developer effort. To address these issues, we also investigated the efficacy of *automatically* selecting the appropriate GC system using *on-line* program behavior.

To enable this, we employed a simple heuristic based on maximum heap size and the heap residency following GC. Given our experience with the annotation-based system, we determined that the best performing collector is consistently GMS for small heaps and SS for large heaps. If the heap availability should change, e.g., to make room for concurrent execution of other programs, our system can automatically switch to GMS or SS accordingly.

In addition to determining what GC system to switch to, we must also identify *when* to switch. Some possible options include heap residency thresholds, GC frequency thresholds, and allocation behavior. As an experiment, we implemented the above heuristic (GMS/SS switching with a 90MB heap size threshold) using a heap residency threshold of 60%. As such, given any application, our system waits until the live data following a collection exceeds 60% of the available heap size. At which point, the system checks whether the maximum heap size is greater than 90MB, and if so, switches to SS; else it switches to GMS. The system uses MS as the initial, default GC system. Our use of a residency threshold enables us to use two different collectors for the two different, commonly occurring, program phases: startup and steady-state.

The primary difference between automatic switching and annotation-guided switching is that the switch occurs *after* program execution has begun. Consequently, there may be optimized methods in the system that are *specialized* for the previous GC system. As such, automatic switching employs both method invalidation and on-stack replacement to invalidate such specialization. Since only very hot methods are optimized by JikesRVM, the number of methods that require invalidation or OSR is small in our experience.

We acknowledge that our automatic GC selection heuristic is simple. We include it as a second example of how our switching framework can be employed; moreover, we provide an empirical analysis of the overheads it imposes in the results section. We intend to study extensively techniques for automatic and adaptive switching as part of future work.

4. EVALUATION

To empirically evaluate the efficacy 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.

4.1 Experimental Methodology

We gathered our results using a dedicated 2.4GHz x86-based single-processor Xeon machine (with hyperthreading enabled) running Debian Linux v2.4.18. We implemented our switching framework within the JikesRVM version 2.2.0 with jlibraries R-2002-11-21-19-57-19.

The standard, best-performing JikesRVM configuration is the *adaptive* configuration. This system identifies "hot" methods (that are then optimized) via samples taken on (timer-based) thread switches. Since this process is non-deterministic, the methods optimized and consequently, execution time across runs (using the same input) is highly variable. To eliminate this non-determinism, we profiled each program off-line 100 times and collected the list of methods selected for optimization by JikesRVM. We then computed the intersection of these files and *annotate* those methods using a system that we developed in prior work [27, 28]. When a method is about to be compiled, the system checks whether the method is annotated, and if so, optimizes it directly. Similar to [34], we refer to this system as *pseudo-adaptive* since only "hot" methods are optimized. Such a system is (more) deterministic and therefore, our experimental measurements contain less variance and can be reproduced directly. The profiles we used are available on our research lab web page [39].

For all of our results, we compiled the JikesRVM boot image with full optimizations. In addition, we inserted write-barriers into all compiled methods and we did not inline allocation sites. This configuration allows us to avoid invalidation and on-stack replacement of boot image methods. However, it also imposes execution overhead in the form of missed inlining opportunities and write-barrier execution for non-generational collectors for all boot image methods that execute. Our results include this overhead. To evaluate the impact of these missed opportunities in general, we present results on the use of invalidation and on-stack replacement for application code at the end of the results section.

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 latter runs do not since all methods have been compiled following the initial invocation. We report results as the average 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 [38], JOlden [13], and JavaGrande [24].

4.2 Results

We next present the empirical evaluation of our system. We evaluate the system using annotation and automatic GC selection.

Annotation-Guided GC Selection

As described in the methodology section, we present performance numbers using the pseudo-adaptive JikesRVM system without compilation overhead. We present the compilation times with our system separately.

For annotation-guided GC selection (described in Section 3), we selected the best-performing GC system for a range of heap sizes by profiling multiple inputs offline (inputs are listed in Table 2). The GCs and switch points that we annotate and use are shown in the same table. For brevity, we present results only for the first input specified in Table 2.

Our system uses the annotation to switch GCs immediately prior to invocation of the benchmark (program load time). Our performance numbers *include* the cost of this switch. In addition, our system compiles methods with the appropriate allocation routine inlined when "hot" methods are fully optimized. The system does

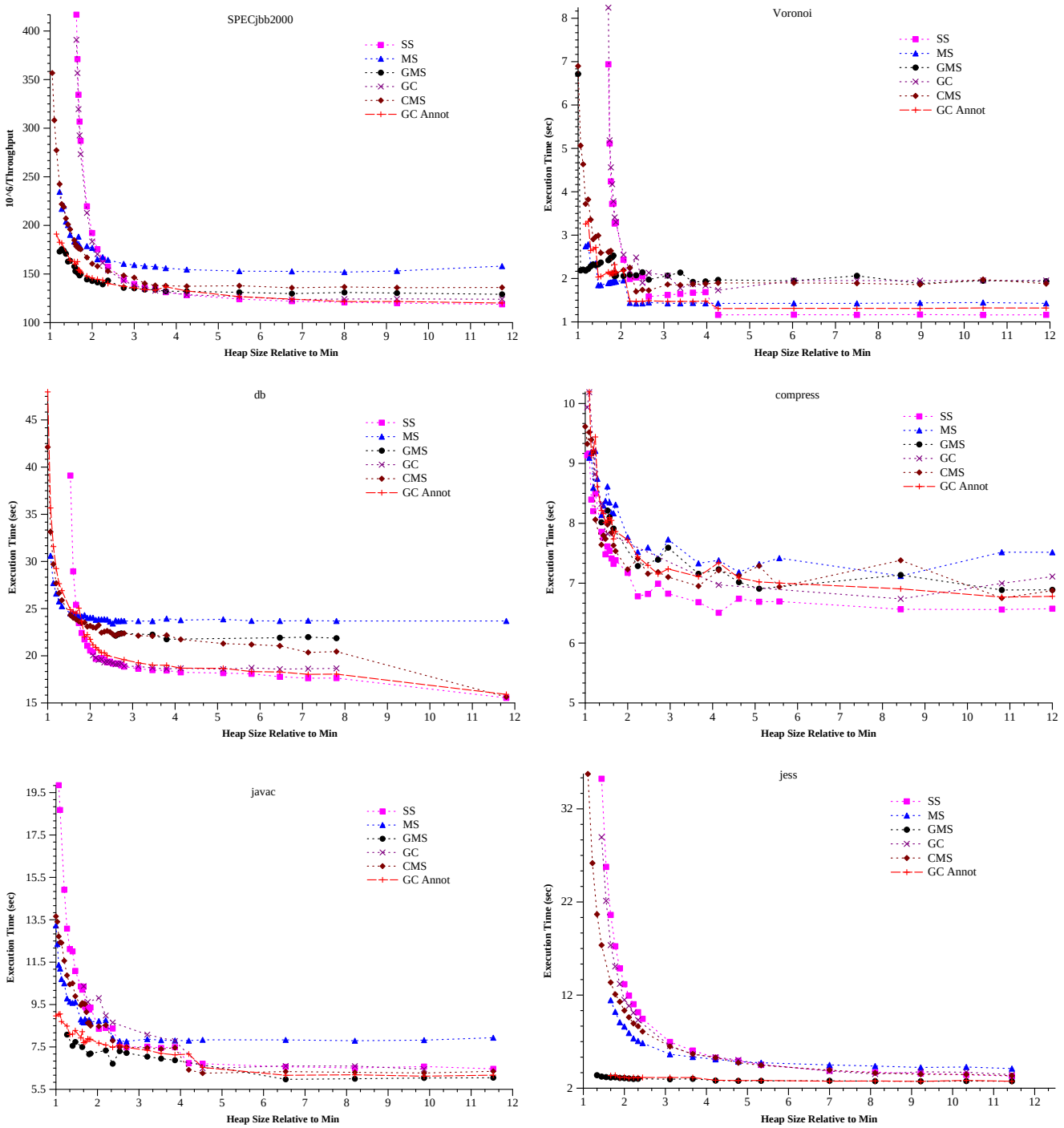

Figure 5: Performance comparison between our switching system, GC Annot (dashed line with + marks), and the unmodified reference system built with five different GC systems. The first three graphs show three representative benchmarks with switch points. The remaining graphs show three representative benchmarks without switch points. The x-axis is heap size relative to the minimum (1 to 12 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/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.

insert write-barriers into all unoptimized (fast-compiled) methods; however, write-barriers are only inserted into optimized ("hot") methods for generational collection systems. Since our system switches to the annotated GC system before the benchmark begins executing, no invalidation or on-stack replacement is required (we discuss the performance impact of employing these mechanisms at the end of the results section).

As we discussed in Section 3, there were 5 of 11 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 for the *same* input.

The first three graphs of Figure 5 shows the results for three representative benchmarks with switch points. The x-axis in each graph is heap size, relative to the minimum (heap size divided by the minimum heap size of the program). We specify the minimum heap sizes per benchmark in column three of Table 2.

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 JikesRVM garbage collectors. These curves represent the performance of the "clean", unmodified, system. The GC systems 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 the plus signs and plotted in red if in color) shows the performance of our GC switching system and annotation-guided selection.

Each of these benchmarks exhibit a switch point (a change in best-performing GC system), and, 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, our system can improve performance over using *any single collector* for these programs.

The rest of the graphs of Figure 5 show the results for three representative benchmarks for which there is no switch point. The layout of the graphs is the same as those described previously. 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 dynamically switch to the best-performing GC for each program, it is able to improve performance across benchmarks over any single GC system. This becomes more evident when we evaluate this data across benchmarks.

The tables in Figure 6 and Table 3 summarize our results across benchmarks and heap sizes. The tables show how our system reduces performance hits taken when the "wrong", i.e., worst-performing collector, is chosen. In addition, they show the average performance degradation over optimal selection. This degradation is due to cross-input differences (for mpegaudio and JavaGrande as explained in Section 3) and to the differences in our system that enable its flexibility, e.g., write-barrier execution in unoptimized code, boot image optimization, switch time (from MS, the default system, to the annotated system), etc.

Table (a) in Figure 6 shows 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 throughput.

Note that the data in these tables do not compare our system against a single JikesRVM GC system; instead, we are comparing our system against the *best- and worst-performing GC system at every heap size*. For example, for large heap sizes for the SPECjbb benchmark, the SS system performs best. For small heap sizes, GMS performs best. In this case, to compute percent degradation, we take 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. On average across benchmarks and heap sizes, our system imposes 4% overhead over the best-performing GC system at each point. Perhaps more importantly however, our system can reduce the overhead of selecting the worst-performing collector (which again may be different across applications and heap sizes) by 26%.

Table (b) presents these same results when we omit Mark-Sweep (MS) collection from consideration. MS works well for small heaps but is thought to implement obsolete technology. As such, we consider the performance of our system without it. On average across benchmarks and heap sizes, our system imposes 3% overhead over the best-performing GC system at each point. In addition, and more importantly, our system can reduce the overhead of selecting the worst-performing collector by 24%. Interestingly, when MS is disregarded from the data, the average degradation actually decreases. This is due to the fact that MS is the best-performing collector for the Voronoi benchmark for small and medium sized heaps. In fact, for this benchmark, the degradation over the best GC is -2%, i.e. an improvement, if MS is not considered. For many of our benchmarks (SPECjbb, db, JavaGrande, etc.), MS is the worst-performing collector, and consequently, the average performance improvement is less than that shown in Table (a).

Finally, Table 3 presents the percent degradation over *always using the Generational/Mark-Sweep Hybrid (GMS)*. GMS is quite popular and thought to be the best-performing, JikesRVM GC system – it is the JikesRVM default collector. GMS exploits the generational hypothesis: programs commonly allocate many, small, short-lived objects. As such, it should be able to perform well in general for a wide range of programs. As the data in Figure 5 shows however, GMS works well in many cases but other GCs enable better performance for some benchmarks (compress, db, Voronoi, etc.). Our system enables a 7% 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 over a 3 second benefit.

Another form of overhead that our system introduces, that is not measured in the previous results, is compilation overhead. On average, our system imposes no significant overhead on fast compilation (0.2%) despite the fact that it inserts write-barriers into all methods. For the optimizing compiler, we introduce 18% overhead (97ms) – since compilation overhead is so small, a small increase in compilation time translates into a large percent overhead. This additional overhead can be attributed to not inlining allocation routines for the boot-image and inserting write barrier checks into the boot-image code (the code that executes when the compilers run).

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 annotated information to guide dynamic switch-

Average Difference Between Best & Worst GC Systems		
	GCAnnot	
Benchmark	Degradation over Best	Improvement over Worst
compress	6.28% (443ms)	3.53% (279ms)
jess	2.82% (85ms)	56.17% (5767ms)
db	2.88% (532ms)	12.47% (3028ms)
javac	5.64% (392ms)	24.12% (2944ms)
mpegaudio	3.54% (214ms)	3.21% (209ms)
mtrt	4.51% (270ms)	42.29% (5170ms)
jack	3.22% (147ms)	32.70% (2787ms)
JavaGrande	3.97% (2511ms)	17.71% (15500ms)
SPECjbb	2.22% ($3.17*10^6$/tput)	27.95% ($82.68*10^6$/tput)
MST	4.07% (30ms)	48.42% (1001ms)
Voronoi	9.20% (144ms)	31.78% (1063ms)
Average	4.38%	26.22%

(a)

Average Difference Between Best & Worst GC Systems		
	GCAnnot	
Benchmark	Degradation over Best	Improvement over Worst
compress	6.28% (443ms)	1.47% (113ms)
jess	2.82% (85ms)	53.23% (5595ms)
db	2.88% (532ms)	8.04% (1866ms)
javac	5.64% (392ms)	21.47% (2727ms)
mpegaudio	3.54 % (214ms)	2.62% (170ms)
mtrt	4.51% (270ms)	42.29% (5170ms)
jack	3.22% (147ms)	30.81% (2681ms)
JavaGrande	3.97% (2511ms)	14.25% (12680ms)
SPECjbb	2.22% ($3.17*10^6$/tput)	24.11% ($76.21*10^6$/tput)
MST	4.07% (30ms)	48.08% (999ms)
Voronoi	-2.00% (-64ms)	31.78% (1063ms)
Geo. Mean	3.36%	24.13%

(b)

Figure 6: Summarized performance differences between our *annotation-guided* switching system and the reference system across heap sizes. Table (a) shows the percent degradation over the best- and percent improvement over the worst-performing GC systems across heap sizes (the time in milliseconds that this equates to is shown in parenthesis). Table (b) is the same as (a) only we have omitted Mark-sweep from the comparison since it is thought to implement obsolete technology.

Benchmark	GC Annot: Average Degradation Over Generational Mark-Sweep
compress	-0.37% (-28ms)
jess	2.82% (85ms)
db	-14.17% (-3122ms)
javac	5.19% (373ms)
mpegaudio	-2.19% (-140ms)
mtrt	2.32% (78ms)
jack	3.22% (147ms)
JavaGrande	-0.19% (-87ms)
SPECjbb	0.95% ($1.72*10^6$/tput)
MST	-44.66% (-827ms)
Voronoi	-11.88% (-241ms)
Geo. Mean	-6.64%

Table 3: Percent degradation of our system over always using the popular GMS collection. The negative values indicate that on average across heap sizes, our system improves performance over GMS.

ing between GC systems. Moreover, when there is a switch point for programs, our system can enable the best performance on average over any single GC system for that program. For cases in which there is no cross-over 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 system across benchmarks.

Automatic Switching

Finally, we empirically evaluated automatic switching (AutoSwitch). To experiment with such a configuration, we implemented the simple heuristic (based on 60% heap residency) described in Section 3.

Invalidation and on-stack replacement are two optimizations that we described in Section 2.2.3 that enable us to aggressively inline allocation sites and avoid inlining write-barriers into hot methods, according to the GC system in use when optimization is performed. As a result, we are able to achieve performance levels similar to that of a non-switching JVM, as shown in the data for annotation-guided GC selection. However, if the system switches GCs during execution of the program, we must *undo* this optimization and replace this code with versions optimized for the GC system to which we have switched.

By employing OSR and invalidation, the compilation system of AutoSwitch has more work to do. It must generate an OSRInfo record for every point in the program at which a GC switch can occur in the JikesRVM, i.e., GC safe-points (call and allocation sites, loop backedges, etc). These records are later discarded once the live variable information is stored into compact OSR maps; however, the cost of their generation and processing is present in the compilation time. On average across benchmarks, our system imposes an additional 348ms in optimization time (for hot methods). AutoSwitch increases fast-compilation time by only 0.2% (0.73ms).

Average Difference Between Best & Worst GC Systems		
	AutoSwitch	
Benchmark	Degradation over Best	Improvement over Worst
compress	8.70% (608ms)	1.71% (140ms)
jess	36.60% (1080ms)	47.05% (5652ms)
db	18.9% (3489ms)	8.54% (2348ms)
javac	26.89% (1817ms)	-4.97% (-367ms)
mpegaudio	12.41% (748ms)	-4.82% (-312ms)
mtrt	30.04% (1816ms)	10.60% (1074ms)
jack	19.29% (855ms)	19.12% (1886ms)
Geo. Mean	15.15%	17.14%

Table 4: AutoSwitch Results. We show the percent degradation over the best- and percent improvement over the worst-performing GC systems across heap sizes (the time in milliseconds that this equates to is shown in parenthesis).

We next measured the performance of AutoSwitch, using the SpecJVM98 benchmarks. Table 4 presents the average performance overhead (without compilation overhead) across these benchmarks and heap sizes (those used in our previous results). The table uses the same format as we used previously for Table (a) in Figure 6. AutoSwitch enables performance improvements over the worst-performing GC of 17% (1488ms) on average. If we do not consider the MS collector, this improvement is 15% (1213ms) on average. However, the average degradation of our system over the best-performing collector is 15% (1489ms). This is significantly worse than annotation-guided GC selection.

The primary difference between annotation-guided GC selection and AutoSwitch is the use of OSR and invalidation of specialized methods. As such, we investigated the overhead imposed by each to identify the source of overhead imposed on AutoSwitch. For small heap sizes, the 60% residency threshold is reached very early during program execution when very few methods have been discovered as hot and optimized. As such, invalidation and OSR overhead, if they occur at all, impose very little overhead (21.0ms on average across benchmarks for invalidation, and 0.0ms for OSR). For

larger heap sizes, this threshold is reached after significantly more execution time has elapsed potentially increasing the number of optimized methods that must be invalidated and, if they are currently executing, OSR'd. For medium-sized heaps invalidation overhead is 25ms on average across benchmarks and for OSR it is 0.6ms. For large-sized heaps, the overhead is 38ms and 5ms on average, for invalidation and OSR, respectively. Our measurements indicate that overall, the overhead introduced by invalidation and OSR is very small for all heap sizes. We have omitted per-benchmark results for brevity; however, they are available in our technical report [37].

Upon further investigation, we discovered the cause of remaining overhead: loss of optimization opportunity. As described in Section 2.2.3, we insert OSRInfo place-holders into the code at all points at which a thread-switch and GC can occur, in order to recover method state during OSR. We implement these OSRInfo points as extensions to the *YieldPoint* instruction, an instruction that causes the currently executing thread to yield the processor to other threads. A fundamental characteristic of yieldpoints in the JikesRVM is that the optimizing compiler *pins* these instructions to ensure that they execute at the point specified; this prevents the optimizer from moving instructions around them (performing code motion).

In our system, unlike the OSR mechanism described in prior work for deferred compilation [15], our OSR points (OSRInfos) are simply markers that are later discarded during code generation. However, the compiler handles them similarly to yieldpoints (which ensures that our OSR maps are correct), and as such, does not perform code motion optimizations across them. In addition, since *OSRInfo* instructions need to use live variables in the program code (similar to *OSRPoint* instructions), other optimizations, like load/store elimination and dead-code elimination are also inhibited. Since there are many points at which OSR *might* occur, the quality of the code that our optimizing compiler produces is significantly worse than that of the clean system (for which there are no OSRInfo points).

We are currently investigating ways to enable these optimizations, regardless of *OSRInfo* points. Specifically, we must incrementally update the OSR map information as optimizations cause the method state (stack/locals) to change. We will investigate the efficacy of this approach as well as other heuristics to guide automatic switching as part of future work.

5. RELATED WORK

Two areas of related work show that performance due to the the GC employed varies across applications and that switching collectors dynamically can be effective. In [30, 32], the authors show that performance can be improved by combining variants of the same collector in a single system, e.g., mark-and-sweep and mark-and-compact. and semispace and slide-compact In [35], the authors show 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 completely unrelated collectors.

Other related work shows empirically that performance enabled by garbage collection is application-dependent. For example, Fitzgerald and Tarditi [16] performed a detailed study comparing the relative performance of applications using several variants of generational and non-generational semispace copying collectors (the variations had to do with the write barrier implementations). They showed that over a collection of 20 benchmarks, 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 prebuilt binaries for each collector, and only examined semispace copying collectors.

Other studies have identified similar opportunities [4, 42, 36]. IBM's Persistent Reusable JVM [21] 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 [7] 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 [7], however, describes the system at a very high level and provides few details or performance data. We were unable to compare our system against the JRockit, due to its proprietary nature. To our knowledge, no extant research has defined and evaluated a general framework for switching between very diverse GC systems, such as the one that we describe. In addition, our automatic switching heuristic, albeit simple, requires no user intervention and achieves considerable performance improvement.

6. CONCLUSION

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 system can be significant. To overcome these limitations, we have developed a framework that can automatically switch between GC systems without having to restart and possibly rebuild the execution environment, as is required by extant systems. Our system can switch between collection strategies *while* the program is executing. Our empirical evaluation shows that the annotation-guided switching system we describe degrades performance by under 4% on average over the best-performing collection system for a particular heap size given the range of heap sizes studied. In addition, our system significantly improves performance (over 24% on average) over the GC system at each heap size and over always using the popular Generational/Mark-Sweep hybrid (by 7% on average).

As part of future work, we plan to investigate techniques that reduce the overhead of automatic switching, 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 whether it is more effective to switch between fewer, more homogeneous garbage collectors, and how our system adapts to variable workloads in a server environment that runs multiple applications concurrently.

Acknowledgements

We would like to thank the anonymous reviewers for providing useful comments for the final version of this paper. This work was funded in part by NSF grant No. EHS-0209195, and an Intel/UCMicro external research grant.

7. REFERENCES

[1] AIKEN, A., AND GAY, D. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* (May 1998).

[2] ALPERN, B., ET AL. The Jalapeño Virtual Machine. *IBM Systems Journal 39*, 1 (2000), 211–221.

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

[4] ATTANASIO, C. R., BACON, D. F., COCCHI, A., AND SMITH, S. A comparative evaluation of parallel garbage collectors. In *Proceedings of the Fourteenth Annual Workshop on Languages and Compilers for Parallel Computing* (Cumberland Falls, Kentucky, Aug. 2001), vol. 2624 of *Lecture Notes in Computer Science*, Springer-Verlag.

[5] BACON, D. F., ATTANASIO, C. R., LEE, H. B., RAJAN, V., AND SMITH, S. E. Java without the coffee breaks: A non-intrusive multiprocessor garbage collector. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Snowbird, Utah, Jun 2001).

[6] BACON, D. F., FINK, S. J., AND GROVE, D. Space- and time-efficient implementation of the Java object model. In *Proceedings of the Sixteenth European Conference on Object-Oriented Programming* (Málaga, Spain, June 2002), B. Magnusson, Ed., vol. 2374 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 111–132.

[7] BEA SYSTEMS INC. BEA Weblogic JRockit: Java for the enterprise. `http://www.bea.com/content/news_events/white_papers/BEA_JRockit_wp.pdf`.

[8] BLACKBURN, S., CHENG, P., AND MCKINLEY, K. 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 2003. `http://eprints.anu.edu.au/archive/00001986/`.

[9] BLACKBURN, S., AND MCKINLEY, K. In or out? putting write barriers in their place. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)* (2002).

[10] BLACKBURN, S., MOSS, J., MCKINLEY, K., AND STEPHANOVIC, D. Pretenuring for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (Tampa, FL, Oct 2001).

[11] BLACKBURN, S. M., JONES, R., MCKINLEY, K. S., AND MOSS, J. E. B. Beltway: Getting around garbage collection gridlock. In *Proceedings of PLDI'02 Programming Language Design and Implementation* (June 2002).

[12] BRECHT, T., ARJOMANDI, E., LI, C., AND PHAM, H. 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'01)* (Nov. 2001).

[13] CAHOON, B., AND MCKINLEY, K. Data Flow Analysis for Software Prefetching Linked Data Structures in Java Controller. In *International Conference on Parallel Architectures and Compilation Techniques* (Sept. 2001).

[14] CLINGER, W., AND HANSEN, L. T. Generational garbage collection and the radioactive decay model. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation* (May 1997), pp. 97–108.

[15] FINK, S., AND QIAN, F. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *International Symposium on Code Generation and Optimization (CGO)* (Mar. 2003).

[16] FITZGERALD, R., AND TARDITI, D. The case for profile-directed selection of garbage collectors. In *Proceedings of the second international symposium on Memory management* (2000), ACM Press, pp. 111–120.

[17] HEWLETT-PACKARD COMPANY. NonStop Server for Java Software. Project home page. `http://nonstop.compaq.com/view.asp`.

[18] HICKS, M., HORNOF, L., MOORE, J., AND NETTLES, S. A study of large object spaces. In *ISMM98* (Mar. 1999).

[19] HIRZEL, M., DIWAN, A., AND HERTZ, M. Connectivity-based garbage collection. In *ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)* (Anaheim, CA, Nov. 2003), SIGPLAN Notices, Association for Computing Machinery.

[20] HUMMEL, J., AZEVEDO, A., KOLSON, D., AND NICOLAU, A. Annotating the Java Bytecodes in Support of Optimization. *Journal of Concurrency:Practice and Experience 9*, 11 (Nov. 1997), 1003–1016.

[21] IBM CORPORATION. Persistent Reusable JVM. Project home page. `http://www.haifa.il.ibm.com/projects/systems/rs/persistent.html`.

[22] IBM CORPORATION. WebSphere software platform. Product home page. `http://www-3.ibm.com/software/info1/websphere/index.jsp`.

[23] INC., S. M. The Java Hotspot Virtual Machine White Paper. `http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_%4_30_01.html`.

[24] Java Grande Forum. `http://www.javagrande.org/`.

[25] JONES, J., AND KAMIN, S. Annotating Java Class Files with Virtual Registers for Performance. *Journal of Concurrency: Practice and Experience 12*, 6 (May 2000), 389–406.

[26] JONES, R. E. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[27] KRINTZ, C. Coupling On-Line and Off-Line Profile Information to Improve Program Performance. In *International Symposium on Code Generation and Optimization (CGO)* (Mar. 2003).

[28] KRINTZ, C., AND CALDER, B. Using Annotation to Reduce Dynamic Optimization Time. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (June 2001), pp. 156–167.

[29] KRINTZ, C., GROVE, D., SARKAR, V., AND CALDER, B. Reducing the overhead of dynamic compilation. *Software: Practice and Experience 32*, 8 (2000), 717–738.

[30] LANG, B., AND DUPONT, F. Incremental incrementally compacting garbage collection. In *Proc. of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques* (St. Paul, Minnesota, 1987), pp. 253–263.

[31] POMINVILLE, P., QIAN, F., VALLEE-RAI, R., HENDREN, L., AND VERBRUGGE, C. A Framework for Optimizing Java Using Attributes. In *Proceedings of IBM Centre for Advanced Studies Conference (CASCON)* (2000), pp. 152–168. `http://www.sable.mcgill.ca/publications`.

[32] PRINTEZIS, T. Hot-swapping between a mark&sweep and a mark&compact garbage collector in a generational environment. In *Usenix Java Virtual Machine Research and Technology Symposium* (Monterey, California, Apr. 2001).

[33] ROSEN, M. BEA's enterprise platform. IDC white paper sponsered by BEA. `http://www.bea.com/framework.jsp`.

[34] SACHINDRAN, N., ELIOT, J., AND MOSS, B. Mark-copy: Fast copying gc with less space overhead. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications* (2003), ACM Press, pp. 326–343.

[35] SANSOM, P. Combining single-space and two-space compacting garbage collectors. In *Proceedings of the 1991 Glasgow Workshop on Functional Programming* (Portree, Scotland, 1992), R. Heldal, C. K. Holst, and P. Wadler, Eds., Workshops in Computing, Springer-Verlag, pp. 312–323.

[36] SMITH, F., AND MORRISETT, G. Comparing mostly-copying and mark-sweep conservative collection. In *Proceedings of the first international symposium on Memory management* (1998), ACM Press, pp. 68–78.

[37] SOMAN, S., KRINTZ, C., AND BACON, D. F. Dynamic Selection of Application-Specific Garbage Collectors. Tech. Rep. 2004-09, Univ. of California, Santa Barbara, Jan 2004. `http://www.cs.ucsb.edu/~ckrintz/abstracts/annotgc.html`.

[38] Standard performance evaluation corporation (SpecJVM98 and SpecJBB Benchmarks). `http://www.spec.org/`.

[39] UCSB RACELAB: The laboratory for Research on Adaptive Compilation Environments. `http://www.cs.ucsb.edu/~racelab`.

[40] UNGAR, D. Generation scavenging: A non-disruptive high performance storage recalamation algorithm. In *Proceedings of the ACM SIGSOFT/SIIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburg, Pennsylvania, Apr 1992).

[41] UNGAR, D., AND JACKSON, F. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems 14*, 1 (1992), 1–27.

[42] ZORN, B. Comparing mark-and sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM conference on LISP and functional programming* (1990), ACM Press, pp. 87–98.