Adaptive, Application-Specific Garbage Collection

UCSB Technical Report #2003-07 March 21, 2003

Sunil Soman
Computer Science Department
University of California, Santa Barbara
Santa Barbara, CA 93106
sunils@cs.ucsb.edu

Chandra Krintz
Computer Science Department
University of California, Santa Barbara
Santa Barbara, CA 93106
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

In this paper, we describe a novel execution environment that can dynamically switch between garbage collection systems. As such, it enables selection of the most appropriate allocator and collector for a given application and underlying resource availability. Our system is novel in that it is able to switch between a wide range of diverse collection systems. It uses program annotations to guide selection of the collection system. In addition, it can automatically identify when to switch collectors when program execution behavior warrants it, i.e., it is adaptive. Our system introduces little overhead and accurately identifies the best collector for a wide range of benchmarks and heap sizes.

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.

The performance of heap allocation and collection techniques has been the focus of much research [10, 12, 9, 16, 1, 13, 31, 6]. The goal of most of this prior work has been to provide general-purpose mechanisms that enable high-performance execution across all applications. However, other prior research [5, 14, 33], 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 as confirmed these findings and, as such, we believe that to achieve the best performance, the collection and allocation algorithms used should be specific to both application behavior and heap size.

Existing execution environments enable application- and heap-specific garbage collection, through the use of different configurations (builds) of the execution environment. However, such systems do not lend themselves well to the next-generation of high-performance server systems in which a single execution environment executes continuously while multiple applications and code components are uploaded by users [32, 24, 19, 23, 15, 20]. For these systems, a single collector and allocator must be used for a wide range of available heap sizes and of applications, e.g., e-commerce, agent-based, distributed, collaborative, etc. As such, it may not be possible to achieve high-performance in all cases given only a single garbage collection system.

In this work, we present the design, implementation, and evaluation of a virtual machine that can dynamically switch between different garbage collectors. This dynamic switching functionality enables us to use the collector and allocator that will provide the best performance for the executing application and underlying resource availability. In addition, we can switch to a new collection system at any time during an application's lifetime. That is, our system can dynamically *adapt* to changes in execution behavior or the operating environment. If the memory usage characteristics no longer warrant the use of the current collector, we switch to another to improve performance. Similarly, if the amount of memory available to the virtual machine increases or decreases because of changes in resource utilization by external processes, we can change garbage collectors accordingly.

We implemented three mechanisms in our system to guide switching decisions. The first is through the use of off-line execution and measurement of the application using multiple inputs. We then annotate the program to indicate which collection system to use given different resource constraints. Our second approach to collector discovery incorporates light-weight, on-line profiling to guide selection. As the program executes, we monitor allocation and collector behavior and switch, i.e., it *adapts*, to a new memory management strategy when our measurements exceed certain thresholds. Finally, we supply the user with a library call so that users can initiate switching manually from within an application.

We implemented our techniques as an extension to an an existing, high-performance, server-oriented Java virtual machine: The Jikes Research Virtual (JikesRVM) [2] from IBM T.J. Watson Research Center. Our system is able to switch between four existing, yet very different and efficient collection systems. In addition, our system can be easily extended to include any number of other collection systems.

Our system accurately identifies the best collector for a wide range of benchmarks and heap sizes. Our empirical evaluation indicates that the cost of switching is equivalent to a garbage collection. In addition, the overhead that our system imposes on application execution performance is small: Our adaptive garbage collection switching system, degrades performance by 5%, on average, for the programs that we studied. In addition, and perhaps more importantly, our system significantly reduces the negative impact of selecting the “wrong” collector (by 17% on average).

This paper makes the following contributions:

- It describes a technique by which a virtual machine can efficiently switch between different collectors at run-time, and provides empirical measurements of our implementation of this technique, which show that it performs well.
- It shows that dynamic switching of collectors can be combined with off-line profiling and explicit annotation-guided collector selection to significantly improve performance by adapting to available resources.
- It shows that dynamic, on-line profiling can be used to adapt to changes in available resources or to phase shifts in the application, leading to significantly increased performance.
- It shows that even using a simple on-line heuristic, dynamic collector selection can achieve performance that is always very close to that achieved by the best possible collector for the available heap size.

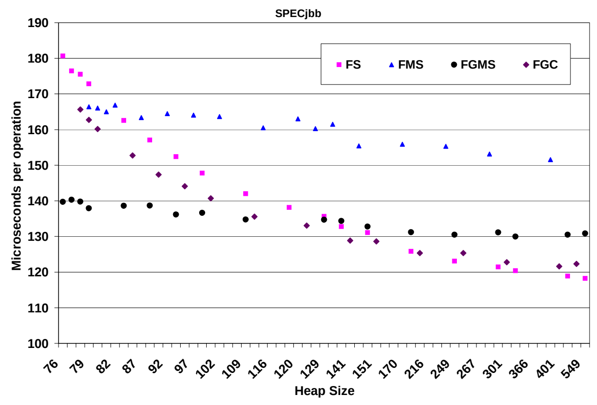
The rest of this paper is organized as follows: Section 2 describes our system for dynamically switching between collectors. Section 3 describes the different approaches we employ to dynamically switch between collectors. Section 4 presents and discusses our experimental results. Section 5 describes potential optimizations that are suggested by the experimental evaluation that should further improve performance. Finally, Section 6 discusses related work and is followed by our conclusions.

2. APPLICATION-SPECIFIC GARBAGE COLLECTION

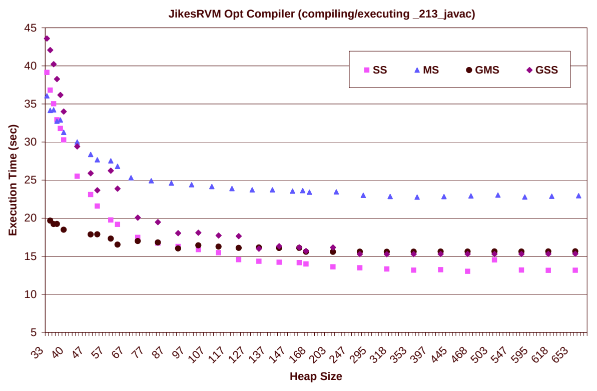
The next-generation of high-performance server systems must enable continuous availability and high-performance if they are to gain wide-spread use and acceptance. As such, these systems require a software execution environment that is also continuously available. Due to the portability, flexibility, 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 [32, 24, 19, 26]. 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 Java execution environments, a single, general-purpose collector and a single allocation policy must be selected and used for all applications. However, many researchers have shown that there is no single combination of a collector and an allocator that enables the best performance for all applications on all hardware and given all resource constraints [5, 14, 33]. Figure 1 confirms these findings. The graphs show the total execution performance of three commonly used SPEC [29] benchmarks executed within the Jikes Research Virtual Machine (JikesRVM) [2]. The y-axis is total time in seconds and the x-axis is heap size in megabytes (MB). For SPECjbb, the y-axis is the inverse of the throughput reported by the benchmark; we report microseconds per operation to maintain visual consistency with the execution time data of the other benchmarks. For the data in all graphs, lower values are better.

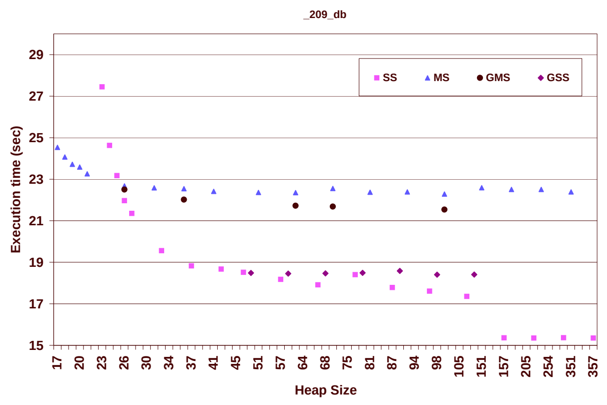
Figure 1(a) shows that for SPECjbb, the semispace (SS) collector, performs best for all but small heap sizes, for which the hybrid generational/mark-sweep (GMS) collector is best. In Figure 1(b),



(a) Heap size versus execution time for SPECjbb



(b) Heap size versus execution time for the JikesRVM optimizing compiler (compiling and executing _213_javac)



(c) Heap size versus execution time for SpecJVM _209_db

Figure 1: Benchmark performance using different collection systems and heap sizes in the JikesRVM. No single collector enables the best performance for all programs and all heap sizes. The collection systems that we used for these experiments are Mark-sweep (MS), Semispace (SS), Generational Semispace (GSS), and Hybrid Generational/Mark-sweep (GMS). The y-axis is total time in seconds. For SPECjbb, the data is the inverse of the throughput reported by the benchmark; we report microseconds per operation to maintain visual consistency with the execution time data. The x-axis is heap size in MB. The benchmarks are from the SpecJVM98 and SpecJBB suites.

GMS is the best for a large number of small to medium sized heaps. Semispace achieves similar performance to that of GMS for large heap sizes. In Figure 1(c), Semispace performs best for heap sizes greater than 26MB. Thereafter, MS outperforms all others. We refer to the point at which the best-performing collection system changes as the *switch point*.

To enable different allocation and collection algorithms to be used for different applications and heap sizes, while enabling continuous execution of the VM, we have developed an execution environment that can dynamically switch between such algorithms. Our goal is to improve performance of applications for which there exist collection system switch points while not imposing significant overhead. Our system is an extension of the JikesRVM from the IBM T.J. Watson Research Center.

2.1 The JikesRVM and the JMTk

The JikesRVM [2] is an open-source, dynamic and adaptive optimization system for Java that was designed and continues to evolve with the goal of enabling high-performance in server systems. The JikesRVM 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, it 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 re-optimization of methods.

The JikesRVM (version 2.2.0+) implements the Java Memory Management Toolkit (JMTk) that enables garbage collection and allocation algorithms to be written and "plugged" into the JikesRVM. The framework offers a high-level, uniform interface to the JikesRVM that is used by all algorithm implementations. Throughout this paper, we refer to the combination of an allocation policy and a collection technique as a *collection system*. This corresponds to a *Plan* in the JMTk terminology. The JMTk allows users to implement their own collection systems easily within the JikesRVM and to perform an empirical comparison with other existing collectors and allocators. When a user builds a configuration of the JikesRVM, she is able to select a particular collection system for incorporation into the JikesRVM image. The selected collection system is specified as a command-line argument to the JikesRVM build process.

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

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

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

In the mark-sweep (MS) collection system, memory is allocated from the mark-sweep space using free-list allocation, like that for large object allocation. Collection is a two-phase process that consists of a mark phase in which live objects are marked, and a sweep phase in which unmarked space is reclaimed.

The generational semispace (GSS) collection system makes use of well-known generational garbage collection techniques [3, 30]. Young objects are allocated in a variable-sized nursery space using bump-pointer allocation. Upon a minor collection, the nursery is collected and the survivors are copied to the mature space. Old objects in the mature space are collected by performing a semispace copying collection. The mature space thus consists of a from-space and a to-space. Objects promoted from the nursery are copied to the from-space. The mature space is collected following a minor collection as needed. This process is referred to as a major collection.

The generational mark-sweep (GMS) collection system also employs a generational model. As for GSS, there is a nursery which holds young objects and a mature space which retains old objects. However, the mature space is collected using a mark-sweep algorithm and allocated using free-list allocation. In earlier versions of the JikesRVM, this collector was previously referred to as the Watson hybrid collector. On a minor collection, surviving objects from the nursery are copied to the mature space. A major collection consists of a minor collection followed by a mature space mark-sweep collection.

The MS and GMS systems do not use a large object space by default. All collection 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.

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

2.2 Switching Between Garbage Collection Systems

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

Our extension to the JikesRVM requires that multiple collection systems be included in a single system image. To enable this, we implemented a generic *Plan* class, from which all specific collection system classes derive, e.g., SSPlan, MSPlan, GMSPlan, and GSSPlan. Each of these plans are instantiated in a single image of our system. Figure 2 shows the JikesRVM JMTk class hierarchy before and after our extensions.

We inserted a global field called *currentPlan* into the class that implements the collection system interface to the JikesRVM (*VM_Interface*). This field identifies the collection system that is currently

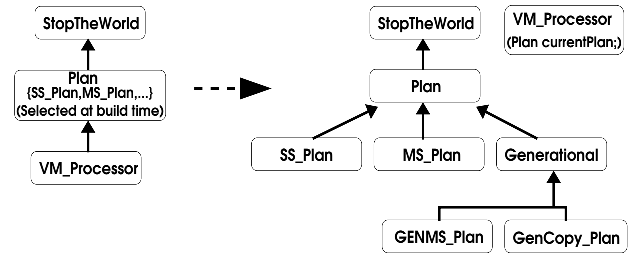


Figure 2: Original and New (Dynamic Switch-Enabled) JikesRVM JMTk Class Hierarchy

in use. At all times, an object instance of each collection system (a plan instance) is available in our system. The current default allocator that is used, depends on the current collection system. The Plan class invokes a static allocation routine according to the collection system indicated by *currentPlan*. When a switch occurs, the instance of the appropriate collector becomes the global instance of the system.

The immortal space and large object space within our system are shared across all collection systems. Since the extant versions of MS and GMS do not implement a large object space by default, we extended both to do so. To support multiple collection systems, we require address ranges for all possible virtual memory resources to be reserved. We try to make efficient use of the virtual address space and to overlap as many address ranges as possible (Figure 3). Note that these address ranges are mapped to physical memory lazily (as it is needed), in 1 Megabyte chunks.

Switching between collection 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. The JMTk *VM_Handshake* class implements synchronization across collector threads and suspension of all mutator threads. While mutators are executing, the collector threads are blocked on the *collectorQueue*. The collector threads are dequeued prior to the start of a collection. The collector threads then perform the actual collection work. When a collector switch is requested, we perform these steps then switch from the old collection system to the new system.

Our implementation, however, does not require garbage collection to be performed for all switches. For example, a switch from MS to GMS or SS to GSS only requires that future allocations are from the nursery area. As such, we only need to perform general book keeping to record the current plan and no collection needs to be performed. Similarly, when we switch from a generational collector to a non-generational collection, we need only perform a minor collection in most cases. After the switch, we suspend collector threads and resume the mutators, similar to the post processing that happens after a regular collection.

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

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

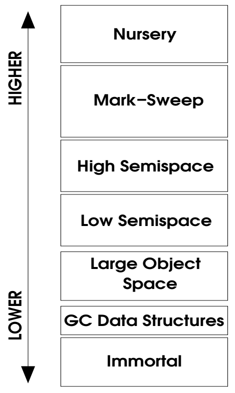


Figure 3: Address Space Layout in the Switching System

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

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

Mark-sweep or Generational Mark-sweep to Semispace. For the MS to SS switch, as we mark live objects in the mark-sweep space, we forward them to the semispace resource. However, this switch is more complex than the ones previously described. The use of object forwarding during a Mark-sweep collection requires us to maintain multiple states per object. We detail this process and its implementation in Section 2.3.

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

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

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

Generational Copying to Mark-sweep/Generational Mark-sweep. This switch is similar to that of SS to MS or GMS. However, we need to copy over objects from the nursery to the shared free-list

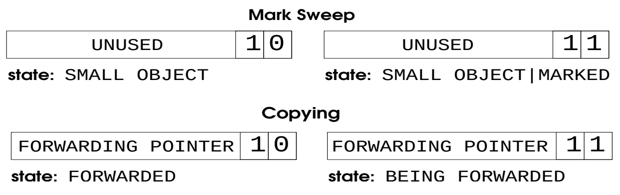


Figure 4: Examples of Bit Positions in the Object's Status Word

region, in addition to copying objects from the GSS mature space to the shared region.

Unlike previous work, we are thus able to dynamically switch between collection systems that use very different allocation and collection strategies. We next describe details that are specific to our implementation within the JikesRVM.

2.3 Implementation Details

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

As mentioned in the previous section, to switch from a collection system that uses a mark-sweep space to a collection system that uses a contiguous semispace, we need to maintain state for the mark-sweep process as well as for the process of forwarding objects to the semispace. In the JikesRVM, the mark-sweep collector requires two bits in the object header: the *mark bit* to mark live objects and the *small object bit* to indicate that the object is a small object. The use of the *small object bit* is specific to the free-list implementation in the JikesRVM. Since memory allocation requests are aligned on a 4-byte boundary in the JikesRVM, the lowest two bits in an object's address are always 0. Hence, the *mark bit* and the *small object bit* can be encoded as the lowest two bits in the object's status word, which is stored in the object header.

The copying process uses two additional states. An object is marked as *being forwarded* while it is being copied. After it is copied, the object is marked as *forwarded* and a forwarding pointer to the object's new location is set in the old object's header. The *being forwarded* state is necessary to ensure synchronization between multiple collector threads. These two states are stored in the status word of each object, along with the forwarding pointer. Thus, the lowest two bits in the object's status word have different purposes depending on the collector that the JikesRVM is configured with, while building the boot image. For example (Figure 4), if the JikesRVM is built with the mark-sweep collection system, the value `0x2` indicates that the object is a small object. However, if instead, the semispace collector is used, this state indicates that the object has been forwarded to the to-space during a collection. Similarly, if the lowest two bits are set, it signifies that the object is a small object and has been marked live by the mark-sweep collector; the same state indicates to a semispace collector thread that the object is currently being forwarded by another thread.

Since we forward marked objects to the semispace, we need to support all four distinct states, along with the space required for the forwarding pointer. To account for the two additional bits required, we use a technique known as bit-stealing [7]. The object header stores a pointer to its Type Information Block (TIB). In the JikesRVM, TIBs store information about the object's class (including the virtual method table). A TIB is defined to be aligned on a 4-byte boundary. Hence, we can use the two lower-most bits of the word that points to the TIB, to mark the *being forwarded* and *forwarded* states during the copying process.

The second feature we implemented is memory unmapping. The reference JikesRVM implementation uses on-demand memory mapping of the virtual address space. To use physical memory efficiently, we unmap the memory mapped space that we won't use when we switch to a new collection system. For example, when we switch from the SS collection system to the MS collection system, we can free memory mapped to the semispace. However, we do not unmap memory from a region that is shared by the old and the new collection systems.

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

A second source of overhead, other than our inability to inline each of the individual allocation routines, which is necessarily introduced by our system is a universal write barrier. Write barriers are used to record pointers for generational collectors [8, 17] since heap areas are independently collected. Cross-generation (old-to-young) pointers must be tracked so that they can be traced during collection of the young (nursery) heap space. Such tracing avoids collecting objects that are only reachable from the older generations. Since our system can switch at any time to a generational garbage collector, we must always insert write barriers. However to make this process as efficient as possible, we use a single, shared write barrier for all collection systems. In our system, the nursery always occupies the highest virtual address range. Hence, we require only a single check to determine if the young object reference is in the nursery. We inline this check into instructions that store the young object reference into an object field or an array.

We discuss the impact of these different sources of overhead using empirical data in our results section (Section 4). In addition, we offer solutions that reduce both inlining and write-barrier overhead in Section 5.

3. GARBAGE COLLECTOR DISCOVERY

By implementing functionality to switch between collection systems while the JikesRVM is executing, we can now select the "best-performing" collection system for each application that executes using our system. We identify each application-specific collection system using three methods: user initiated switching and off-line and on-line program profiling.

To enable user controlled switching between collection systems, we implemented a library call (*VM_Interface.switchGC(System.GCName)*) that can be invoked from a user program that forces the current collection and allocation strategies to switch from one collection system to another. The call is similar to the currently available *System.gc()* which forces a garbage collection regardless of whether the heap is exhausted. The *System.GCName* parameter represents the new collection system. Upon invocation of *VM_Interface.switchGC(System.GCName)*, the mutators pause, the current collection system changes over the new system, and the mutator threads resume. We next detail our off-line and on-line profile mechanisms.

3.1 Annotation-guided Selection

Our next mechanism for determining when to perform a switch, is application annotation of collection systems as identified by off-line profile information. We annotate programs with collection system identifiers, i.e., those used to invoke the *VM_Interface.switchGC(System.GCName)* library call, that are available within the JikesRVM. If an annotated identifier is not available, the default JikesRVM collection system is used. We specify (possibly multiple) collection systems for a number of heap size ranges. We insert annotations into bytecode programs using an annotation language and a highly compact encoding that we developed in prior work [21].

We discover the best-performing collection systems by repeatedly executing the program off-line for a number of heap sizes and program inputs and recording the systems that enable minimum program execution times. 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 that we describe in Section 4.1 (Experimental Methodology).

If the collection system indicated by our program annotations is incorrect (due to differences in application behavior across inputs), we will identify this case using on-line profiling and automatically correct our choice using adaptive garbage collection.

3.2 Adaptive Garbage Collection

Our final mechanism for collection system discovery is on-line instrumentation and profiling similar to that which is implemented by the JikesRVM adaptive optimization system [4]. We monitor execution behavior while the program is running, to *predict* which collection system will enable the best application performance. We use a simple heuristic in which the system identifies when the heap has reached 60% of its capacity. It then checks the available heap size and if it is larger that 90MB, it switches to GSS. Otherwise it uses GMS. We determine this threshold using empirical data from a large number of benchmarks and inputs. We found that the best performing collector for small heap sizes is consistently GMS. For large heap sizes, the best performing collector is commonly GSS. We plan to consider other heuristics and runtime information that can be exploited to guide selection of a garbage collection system as part of future work. As an initial attempt, our simple heuristic performs quite well (as we will show in Section 4) for the programs that we studied.

Dynamic switching between collection systems also enables us to identify additional switching opportunities. That is, we continue to monitor application execution characteristics to determine how well our initial choice of the collection system performs and to determine when and if will be profitable (in terms of execution performance) to switch again. That is, as the behavior of the application changes, we can switch between collection systems to *adapt* and improve program performance.

3.3 Considering System Characteristics

Using each of the mechanisms above for collection system discovery, we can consider a wide range of resource and system characteristics to identify when a particular system is best. Three such characteristics are heap size, the JikesRVM configuration, and available virtual memory resources.

As shown previously in Figure 1, the best-performing collection

Program	Description
_201_compress	SpecJVM98 compression utility, input 100
_209_db	SpecJVM98 database access program, input 100
_228_jack	SpecJVM98 Java parser generator based on the Purdue Compiler Construction Tool set, input 100
_213_javac	SpecJVM98 Java to bytecode compiler, input 100
_202_jess	SpecJVM98 expert system shell benchmark: Computes solutions to rule based puzzles, input 100
_222_mpegaudio	SpecJVM98 audio file decompression tool that conforms to the ISO MPEG Layer-3 specification, input 100
_228_mtrt	SpecJVM98 multi-threaded ray tracing implementation, input 100
JavaGrande	JavaGrande Forum benchmark suite using input section3/AllSizeA
OptComp	The JikesRVM non-adaptive, optimization system executing benchmark _213_javac, input 100
SPECjbb	Transaction processing application with a single warehouse as input

Table 1: Benchmark Descriptions

system can be different for a single application given different heap sizes. As such, we must identify, for a number of common heap sizes, which collection system to select. The available heap size is dependent upon the underlying system resources (which are highly varied for Internet-computing) as well as on other applications sharing the same resources. For example, if multiple programs have been uploaded and are being executed concurrently within the same Java execution environment, the heap space available for use by a particular application may be significantly less than if the program was executing in isolation.

Memory usage characteristics are also dependent upon the build configuration of the JikesRVM. If the adaptive optimization system (AOS) is in use, only those methods identified as "hot" are compiled with optimization. Hot methods are identified using on-line instrumentation and monitoring of execution performance. When adaptive optimization is not in use (via a non-adaptive configuration), no instrumentation is performed, no additional data structures are needed to log profile data, and all methods are optimized upon first use. Each method optimized uses significant memory for storing analysis data and intermediate representations of the code. Non-adaptive configurations can be used when compilation overhead is not an issue, i.e., the program executes for a long period of time. In our results section, we report our findings and switching results that we obtained using both JikesRVM configurations (with and without compilation).

The collection system configuration also impacts performance. For example, any collector can implement a separate large object space. For some applications, use of a large object space improves performance. For others, it degrades performance. For example using the semispace collector for the SpecJVM98 benchmarks and the default heap size, using a large object space improves performance for _201_compress benchmark by 5% and degrades performance for all other SpecJVM98 and SPECjbb benchmarks by 8% on average. We can annotate this information with each benchmark so that the large object space is used only when we predict, given off-line measurement, that it will improve execution speeds.

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

Switching Cost (ms) (GC cost on diagonal)				
	SS	MS	GMS	GSS
SS	**150**	152	150	52
MS	150	**140**	51	151
GMS	140	20	**19**	170
GSS	19	140	140	**19**

Table 2: Time in milliseconds to perform a switch from a collection system in the first column to the one in the first row. The diagonal shows the time for a collection in the reference system. The heap is empty (1.4MB/500MB heap) when the collection or switching is performed.

Program	Annotated GC Selector
_201_compress	if (heapsize $\geq$ 50MB) GSS else GMS
_209_db	if (heapsize $\geq$ 30MB) SS else MS
_228_jack	GMS for all heap sizes
_213_javac	GSS for all heap sizes
_202_jess	GMS for all heap sizes
_222_mpegaudio	SS for all heap sizes
_228_mtrt	if (heapsize $\geq$ 40MB) GSS else GMS
JavaGrande	if (heapsize $\geq$ 72MB) GSS else GMS
OptComp	if (heapsize $\geq$ 118MB) SS else GMS
SPECjbb	if (heapsize $\geq$ 150MB) SS else GMS

Table 3: Annotated garbage collection selection decisions for each benchmark

these benchmarks and our experimental methodology with which we generated our results.

4.1 Experimental Methodology

We gathered the results that follow by repeatedly executing benchmark programs on a dedicated 2.4GHz x86-based single-processor Xeon machine (with hyperthreading enabled) running Debian Linux v2.4.18. In addition, we used the JikesRVM version 2.2.0 with jlibraries R-2002-11-21-19-57-19. We experimented with both the Fast and Adaptive JikesRVM configurations. In the Fast configuration, all methods executed are optimized upon initial invocation. With the Adaptive configuration, all methods are initially compiled without optimization and then monitored (instrumented and profiled) to identify methods that account for a large portion of execution time. These "hot" methods are then optimized. We report numbers using the Fast configuration for brevity; however, the impact of dynamic switching is similar for either system.

We measured the impact of switching both on application performance alone and application performance with the JikesRVM compilation overhead. To measure the impact of switching on application performance in isolation, 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. To evaluate the performance impact of our system on the JikesRVM compiler itself, we also use it as one of our benchmarks. The other applications we examine are from the SpecJVM98, the SPECjbb200, and the JavaGrande [18] benchmark suites. We provide a description of the benchmarks and the inputs that we used in this study in Table 1.

4.2 Results

The first set of results that we present show the overhead of switching alone. We compare the time for a switch with the time for a collection in the baseline system. We performed these experiments with an empty heap (1.4MB of data in a 500MB heap); this enables us to isolate the overhead due to switching. Table 2 shows these results. The data in the table shows the time in milliseconds to perform a collection system switch. A switch is made from the collection system of a given row to that of a column. The diagonal shows the time for a collection in the reference system. The collection performed by the generational collectors (GMS and GSS) is a minor collection in which only the nursery is collected. These results indicate that the cost of a switching is very similar to the cost of a collection.

We next present results that show the efficacy of switching. The results are shown in Figures 5 and 6 for a number of benchmarks.

Benchmark	Compilation Times (sec)			
	input1		input2	
	Reference	Switch	Reference	Switch
_201_compress	1655	1302	1655.5	1306.5
_202_jess	3572	2781	3595.5	2822.5
_209_db	1906	1508	1891	1483
_213_javac	6210	5671	6313.5	5877
_222_mpegaudio	2349.5	2067.5	2346.5	2052.5
_227_mtrt	2252.5	2008	2256.5	2011
_228_jack	6347	4211	6343	4211.5
JavaGrande	2758	2631.5	2682.5	2628
SPECjbb	13014	8566.5	12978	8594.5

Table 4: Compilation overhead introduced by JikesRVM dynamic compilation and optimization.

The x-axis is heap size in megabytes and the y-axis is total time (in milliseconds) for program execution. For SPECjbb, the y-axis is microseconds/operation which is the inverse of the throughput reported by the benchmark; we report this metric to maintain visual consistency with the execution time data, i.e., lower numbers are better.

Each graph shows the performance of each of the garbage collection systems that we studied: MS, SS, GSS, and GMS. In addition, each shows the performance of our annotation-guided (+ GCAnnot) and adaptive (X GCAdapt) collection system. The results indicate that our system is able to track the best performing collector and switch to it. For cases in which there is no cross-over between collectors, e.g., _201_compress, _202_jess, _228_mtrt, _213_javac, our system maintains performance similar to that of the reference system.

For annotation-guided collector selection (GCAnnot), we ran the programs off-line using a number of inputs and identified what collectors to use for different heap sizes. The collectors chosen are listed in Table 3. For adaptive selection, we use the heuristic described previously: For heap sizes larger than 90MB, the GSS collection system is switched to, once residency exceeds 60%. If the heap size is less than 90MB, then GMS is used. In general, this simple heuristic works quite well. However, _209_db shows how the performance that results from having perfect information (annotation) diverges from that of adaptive collector selection. The adaptive system tracks GSS, but it is not the best-performing collector in this case. However, the overhead introduced by our system does not cause noticeable degradation over using the GSS collector in the reference system.

The overhead introduced by our system is low for most benchmarks. However, _228_jack shows significantly more overhead due to switching than for other benchmarks. We believe that this is due to loss of inlining opportunities for allocation sites. Since our system must dynamically check the type of collection system in use prior to deciding which allocation routine to invoke, we are unable to freely inline such sites. We discuss other possible sources of as well as solutions to such overhead in the next section.

Interestingly, our inability to inline allocation routines can also have a positive effect on program performance. For some programs, e.g., the JikesRVM Optimizing compiler benchmark, switching enables performance that exceeds that enabled by all reference collectors. The improvement we achieve is due to reduced compilation overhead. The optimizing compiler aggressively inlines methods, including those for allocation. As a result, a large amount of time is spent repeatedly optimizing inlined code. This can be seen in Table 4 which shows the time in milliseconds for compilation in the reference system and our switching system. The overhead of our system is significantly lower for such cases because allocation sites cannot be inlined.

Table 5 shows the average difference between the data presented in the graphs in Figures 5 and 6. Columns two and three show the percent degradation that our GCAnnot (column 2) and GCAdapt (column 4) imposes over the best-performing collection system. In parentheses, we show the average absolute difference in milliseconds; for SPECjbb the value in parenthesis is the inverse throughput difference in microseconds per operation. The third and the fifth columns show similar data for the average percent and absolute improvement over the worst-performing garbage collection system on average, across the heap sizes for which we have GCAnnot and GCAdapt switching data.

Note that the data in this table does not compare our system against a single JikesRVM collection system; instead, we are comparing our system against the best- and worst-performing collection system for individual heap sizes. For example, for large heap sizes for the SPECjbb benchmark, the SS system performs best. For small heap sizes, GMS performs best. To compute the percent degradation that our system imposes for each configuration (GCAnnot and GCAdapt), we compute the performance difference between our system and the SS collection system for large heap sizes, and our system and the GMS system for small heap sizes. We compute the average percent and absolute improvement in the same way; however, we compare our system with the worst-performing collector for individual heap sizes.

In some cases, e.g., _202_mpegaudio, GCAdapt performs better than GCAnnot despite the fact that GCAnnot uses perfect information from off-line profiles to guide selection. This is due to differences in the amount of data we collected for each of the configurations as well as to the time at which the switching occurs. For the latter, in the annotation-guided system, when an application is loaded for the first time we perform a switch, possibly requiring a collection. In the adaptive system we wait until the heap reaches 60% capacity before switching. This difference in timing can cause both the application and the garbage collection process to behave differently resulting in different timings for different heap sizes.

In columns two and four (Degradation over Best), some values are negative. In these cases, the performance of our switching system is better than that of of all JikesRVM reference configurations on average, i.e., our system improves performance over the best-performing collection system. As we described previously in this section and in Section 2.3, we believe that this is due to the combined effect of missed inlining of allocation sites and to our use of static collection system method invocations that avoid dynamic dispatch used in the reference system. We are currently investigating the individual impact of each of these differences, and plan to report on them in the final version of this paper.

Our annotation-guided switching system, GCAnnot, degrades performance of the best-performing collection system by 0.31% on average across benchmarks. In addition, this system improves performance by 19% over the worst-performing JikesRVM garbage collection system. The adaptive switching configuration, GCAdapt, degrades performance of the best-performing collector by 5% on average. GCAdapt achieves an improvement of over 17% on average. That is, our system imposes very little overhead (5%), even when we do not have perfect information (profile data communicated via annotations) to guide selection of the best-performing garbage collector. In addition, our system can significantly reduce (17%) the negative impact of selecting the "wrong" collector for a given application and heap size.

5. OPTIMIZATIONS

In the work presented so far, we are able to dynamically switch between collectors to achieve performance that is always within a few percent of the performance achieved with the best-performing collector. However, we would like to avoid paying even this small penalty.

The key to achieving this goal is to use the adaptive optimization subsystem of the JikesRVM [4]. The performance penalties we incur are due to two sources: write barriers which are not needed by all collectors, and loss of inlining due to dynamic dispatch to allocation code.

In the adaptive system, all methods are initially compiled without optimization (using the "baseline" compiler). Only when a method is discovered to be "hot" is it compiled with the optimizer. As a result, the vast majority of all methods are never compiled with the optimizer.

Therefore, we can specialize the optimized methods for the currently running collector, and on switching collectors we can if necessary invalidate the optimized code and recompile it incrementally. While we will pay a small performance cost for the recompilation, this should be easily amortized by any but very short-running applications.

On the other hand, the baseline compiler will continue to compile methods in a generic fashion that will allow them to run with any of the collectors (that is, with write barriers always included and dynamic dispatch to the appropriate allocation routine). Therefore, the majority of the methods (those compiled with the baseline compiler) will not need to be recompiled on a collector switch.

5.1 Write Barriers

Currently, the GMS and GSS collectors share the same generational write barrier, while the MS, and SS collectors do not require a write barrier. Therefore, when running with MS or SS collection systems, the optimized code can be generated without write barriers. On a switch to GMS or GSS, the optimized code must be invalidated and will eventually be recompiled as it is once again determined to be "hot".

When running GMS or GSS, the optimized code must be compiled with write barriers. However, on a switch to MS or SS, the optimized code can be invalidated lazily: the presence of the write barriers in the optimized code is wasteful, but will not cause a correctness problem.

5.2 Inlining of Allocation Methods

The GMS, GSS, and SS collectors all share a common bump-pointer contiguous allocation method. Therefore, we can switch

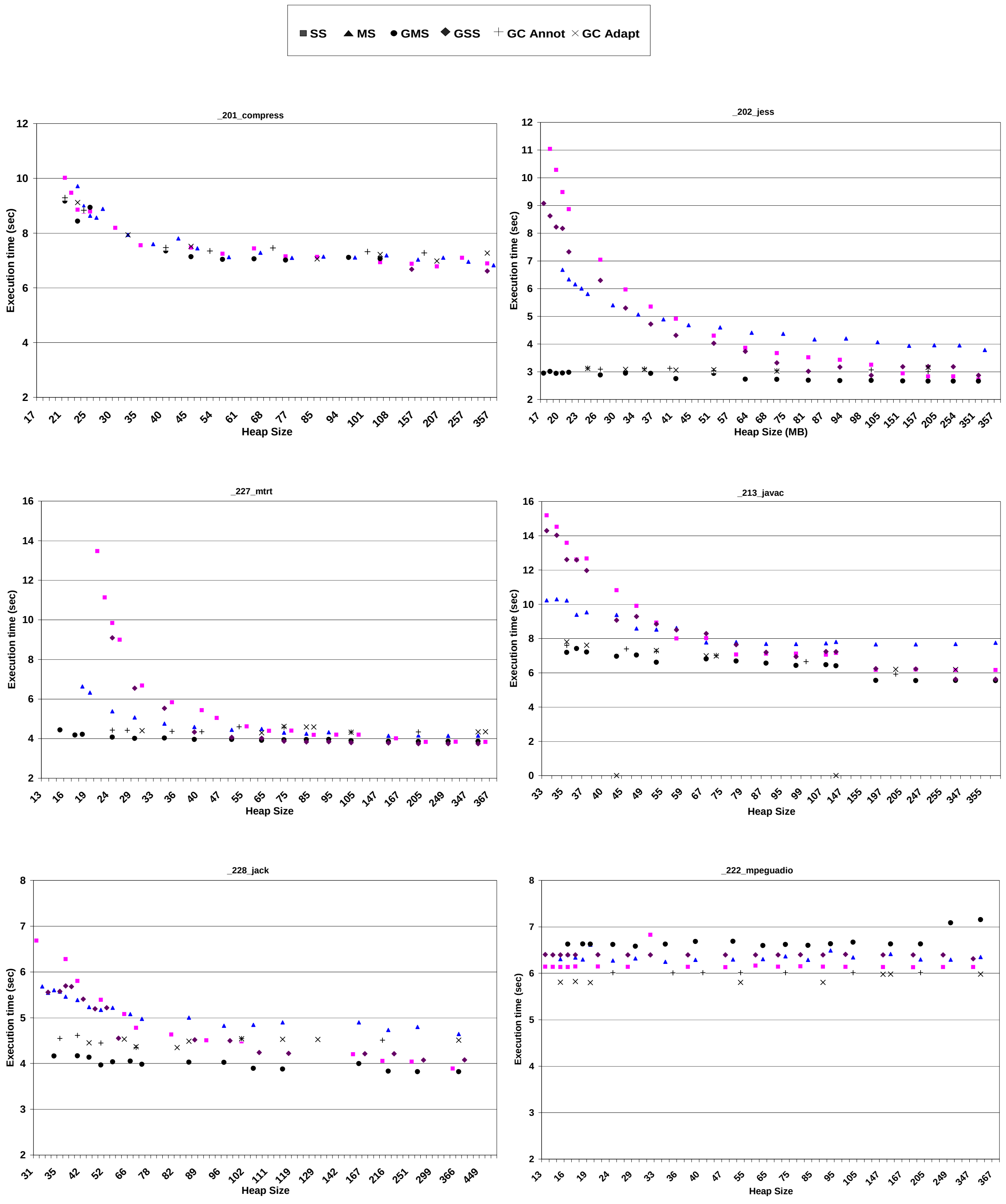


Figure 5: Performance Results. For each benchmark, data for the reference system is shown (SS, MS, GMS, GSS). In addition, the data demarked with (+) and (x) show the efficacy of annotation-guided garbage collection system selection and adaptive garbage collection, respectively. In all cases, our two mechanisms accurately identify the best collector to use and switches to it.

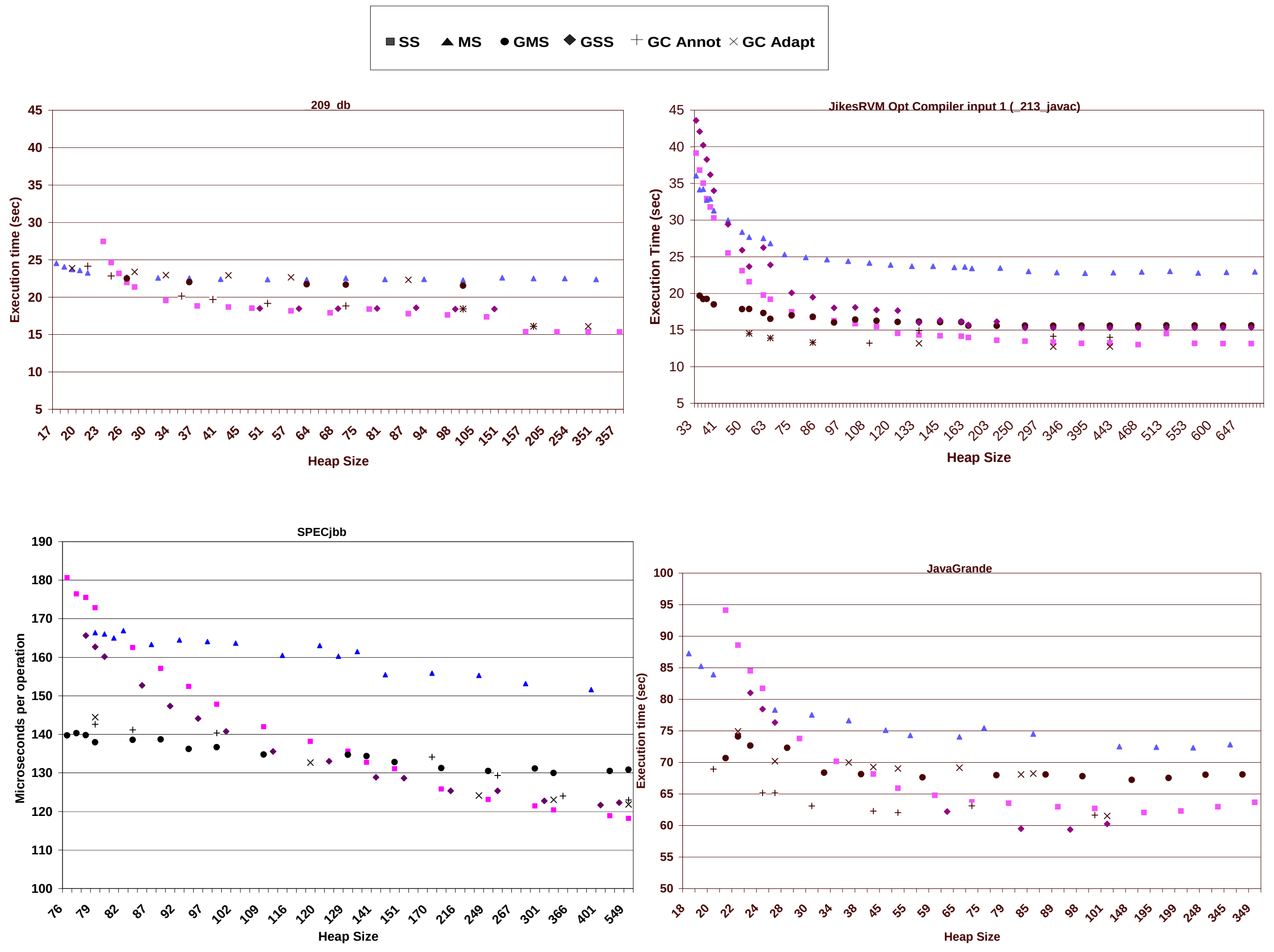


Figure 6: Performance Results, continued. For each benchmark, data for the reference system is shown (SS, MS, GMS, GSS). In addition, the data demarked with (+) and (x), show the efficacy of annotation-guided garbage collection system selection and adaptive garbage collection, respectively. In all cases, our two mechanisms accurately identify the best collector to use and switches to it.

Average Difference Between Best & Worst GC Systems at Each Heap Size Measured				
	GCAnnot		GCAdapt	
Benchmark	Degradation over Best	Improvement over Worst	Degradation over Best	Improvement over Worst
_201_compress	5.52% (378ms)	0.60% (80ms)	7.84% (524ms)	0.97% (99ms)
_202_jess	9.22% (256ms)	38.87% (2220ms)	8.82% (245ms)	39.18% (2212ms)
_209_db	4.04% (745ms)	12.87% (2906ms)	12.80% (2358ms)	7.27% (1632ms)
_213_javac	5.79% (380ms)	23.50% (2408ms)	7.64% (484ms)	17.40% (2556ms)
_222_mpegaudio	-2.33% (-144ms)	9.53% (633ms)	-13.83% (-849ms)	20.63% (1374ms)
_227_mtrt	12.71% (497ms)	17.40% (1421s)	15.48% (594ms)	1.38% (163ms)
_228_jack	11.88% (476ms)	15.29% (865ms)	12.62% (499ms)	10.32% (528ms)
OptComp	-7.85% (-1388ms)	43.02% (10687ms)	-11.89% (-1939ms)	45.80% (11376ms)
JavaGrande	-7.19% (-5440ms)	17.84% (13916ms)	5.60% (3406ms)	10.45% (81300ms)
SPECjbb	3.63% (4.63 usecs/op)	16.03% (26.45 usecs/op)	1.79% (2.29 usecs/op)	18.41% (29.07 usecs/op)
Average	0.31%	19.49%	4.69%	17.18%

Table 5: Average Performance Differences (Absolute Error) between the Garbage Collection Switching System and the Reference System. Column 1 shows the average percent degradation over the best-performing collection system a each heap size for which we collected GCAnnot (annotation-guided switching) data. In parenthesis is the average absolute difference. Column two shows the average percent improvement over the worst-performing collector at each GCAnnot data point. Columns four and five and show the same data for the adaptive switching system, GCAdapt. On average, our annotation-based and adaptive GC switching system degrades performance by 0.31% and 5%, respectively, on average. In addition, each of these systems significantly reduces (19% and 17%, respectively) the negative impact of selecting the "wrong" collector for a given application and heap size.

between any of these collectors without changing the allocation code. It is only the MS collector which allocates directly into a segregated free list data structure.

Therefore, under any of the bump-pointer allocator collectors, we can generate optimized code which inlines the allocation method. If we switch to the MS collector, any optimized method which performs allocation must be invalidated. Similarly, if we are running under the MS collector, we can inline the allocation method and must invalidate methods that invoke allocation on a switch to any of the other collectors.

5.3 Evaluation

The final paper will include an implementation of the dynamic invalidation technique and measurements of its relative performance improvement, and of how close it comes to achieving the performance of "oracular" collector selection.

6. RELATED WORK

There is a small body of related work both showing that different garbage collectors are better for different applications, and that switching collectors dynamically can be effective.

Lang and Dupont [22] built a hybrid system that combines mark-and-sweep with mark-and-compact collection. The entire heap is collected using mark-and-sweep, and a certain (rotating) region is compacted.

Sansom [27] describes a collector for the Spineless Tagless G-machine that switches between a semi-space copying collector and a sliding-compacting collector. The motivation is to allow the program to continue to run when the heap residency exceeds 50%, which would consume an entire semispace. He describes the trade-off curve between the two curves analytically, and presents some preliminary performance results for synthetic benchmarks showing that the crossover point occurs between 25% and 30% heap residency.

Printezis [25] describes a system that dynamically switches between using mark-and-sweep and mark-and-compact for its old generation (the two collectors share a young generation). The old space is structured in such a way that switching between mark-and-compact and mark-and-sweep does not require reformatting the heap (it just requires creating the free list structures). After growing the heap, the collector switches to mark-and-compact, since it can use the newly obtained free region to perform fast bump-pointer allocation; then it switches back to mark-and-sweep, which has faster old space collection. The collector also switches to mark-and-compact when fragmentation is detected, in an attempt to reduce memory requirements.

Fitzgerald and Tarditi [14] 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 garbage collectors. However, they did not take variations in input, required different prebuilt binaries for each collector, and only examined semispace copying collectors.

Attanasio et al [5] compared the performance of different garbage collectors in the IBM JikesRVM across a variety of heap sizes, and showed that there was significant performance differences across applications and heap sizes. Like Fitzgerald and Tarditi's work, this provides experimental justification for building a system that is capable of switching collectors dynamically. Note that their measurements were for a different set of collector implementations (the "Watson" collectors, as opposed to the "JMTk" collectors measured in this paper), so are not directly comparable to our measurements.

Zorn [33] used traces from medium-sized Common Lisp programs to comparatively evaluate mark-and-sweep and semispace copying collectors, both augmented with multiple generations. He found that the mark-and-sweep collector was slightly slower, but used significantly less memory.

Smith and Morrisett [28] implemented a mostly-copying conservative collector for an SML compiler that compiled to C code, and compared it to the widely used Boehm-Demers-Weiser collector [11]. Their primary goal was to show the efficacy of their new algorithm, rather than to comparatively evaluate the two collectors. For a collection of small ML benchmarks, they found that their algorithm yielded overall application speeds within 20% of those provided by the BDW collector.

7. CONCLUSIONS

Garbage collection is a mechanism that greatly simplifies the program development cycle. In addition, it 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" collection system can be significant. As such, it is important to identify strategies that enable improved performance given a wide range of diverse application domains, e.g., e-commerce, agent-based, distributed, collaborative, etc.

To this end, we present a system that can switch between garbage collection systems without having to restart and possibly rebuild the execution environment (as is required by extant systems). Our system switches collection strategies *while* the program is executing. Our system enables application-specific collection policies that are also specific to the underlying resource availability.

Our implementation is quite flexible and efficient since virtual memory resources are shared across collection systems. In addition, very different collection systems can reside in a single system and can be switched to very efficiently. Users of the system can annotate their programs (or such annotation can be automated) to indicate which collector should be used during execution. Moreover, our system can determine *automatically and accurately* which collector should be used. Our empirical evaluation shows that the switching system we describe degrades performance by only 5% on average over the best-performing collection system for a particular heap size, over the range of heap sizes studied. In addition, our system significantly improves performance (17% on average) over the worst collection system at each heap size. That is, our system automatically avoids "bad" garbage collection system selection decisions.

As part of future work, we plan to investigate other techniques for automatically identifying switch points. 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 identify opportunities in which we can reduce the overhead of switching, e.g., using dynamic recompilation to enable aggressive inlining of allocation sites and more efficient write-barrier implementation.

8. REFERENCES

[1] AIKEN, A., AND GAY, D. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN '98 Con-*

ference 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] ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. Adaptive optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (Oct. 2000).

[5] ATTANASIO, C. R., BACON, D. F., COCCHI, A., AND SMITH, S. A comparative evaluation of parallel garbage collectors. In *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.

[6] 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).

[7] 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.

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

[9] 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).

[10] 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).

[11] BOEHM, H.-J., AND WEISER, M. Garbage collection in an uncooperative environment. *Software – Practice and Experience 18*, 9 (Sept. 1988), 807–820.

[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] 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.

[14] 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.

[15] GRAY, R., KOTZ, D., CYBENKO, G., AND RUS, D. D'Agents: Security in a multiple-language, mobile-agent system. In *Mobile Agents and Security*, G. Vigna, Ed., vol. 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998, pp. 154–187.

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

[17] HOSKING, A. L., MOSS, J. E. B., AND STEFANOVIC, D. A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (1992), pp. 92–109.

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

[19] JRun. Project home page. `http://www.hallogram.com/jrun`.

[20] Jumping Beans. Project home page. `http://www.jumpingbeans.com`.

[21] 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.

[22] 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.

[23] LANGE, D., AND OSHIMA, M. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley Longman, 1998.

[24] NonStop Server for Java Software. Project home page. `http://nonstop.compaq.com/view.asp`.

[25] 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).

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

[27] 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.

[28] 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.

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

[30] 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).

[31] UNGAR, D., AND JACKSON, F. An adaptive tenuring policy

for generation scavengers. *ACM Transactions on Programming Languages and Systems 14*, 1 (1992), 1–27.

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

[33] 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.