Dynamic Selection of Application-Specific Garbage Collectors UCSB TR#2004-09 January, 2004

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 (GC) systems. As such, it enables application-specific GC selection. In addition, the system can switch between different GC systems *while* the program is executing. Our system is novel in that it is able to switch between a wide range of diverse collection systems. To empirically evaluate our system, we implemented annotation-guided GC selection and we show its efficacy for a wide range of benchmarks and heap sizes. In addition, we implemented a simple heuristic that automatically identifies when to switch collectors when program execution behavior warrants it. Our system introduces an average overhead of 4% for both annotation-guided and automatic switching. Perhaps more importantly however, we significantly improve performance over selecting the wrong collection system (by 19% using annotation-guided selection and by 16% using automatic switching, on average).

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, 11, 9, 16, 1, 12, 30, 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, 31, 27], has shown that the efficacy of a memory management system (the allocator and the garbage collector) is dependent upon application behavior and available resources. That is, no single collection system enables the best performance for all applications and all heap sizes. Our empirical experimentation confirms these findings. Over a wide-range of heap sizes and the 10 benchmarks studied, we found that *every* collector enabled the best performance at least once; *including* a mark-and-sweep and non-generational copying collector, two collectors that are commonly thought of as implementing obsolete technology. As such, we believe that to achieve the best performance, the collection and allocation algorithms used should be specific to both application behavior and heap size.

Existing execution environments enable application- and heap-specific garbage collection, through the use of different configurations (via separate builds or command-line options) of the execution environment. However, such systems do not lend themselves well to next-generation, high-performance server systems in which a single execution environment executes continuously while multiple applications and code components are uploaded by users [20, 15, 25] 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 between many different types of collectors, e.g., semi-space, mark-and-sweep, copying-marksweep, and many variants of generational collection.

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

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

This paper makes the following contributions:

- It provides a JVM implementation that is able to dynamically switch between a diverse set of garbage collection systems,
- It shows how dynamic switching can be successfully guided

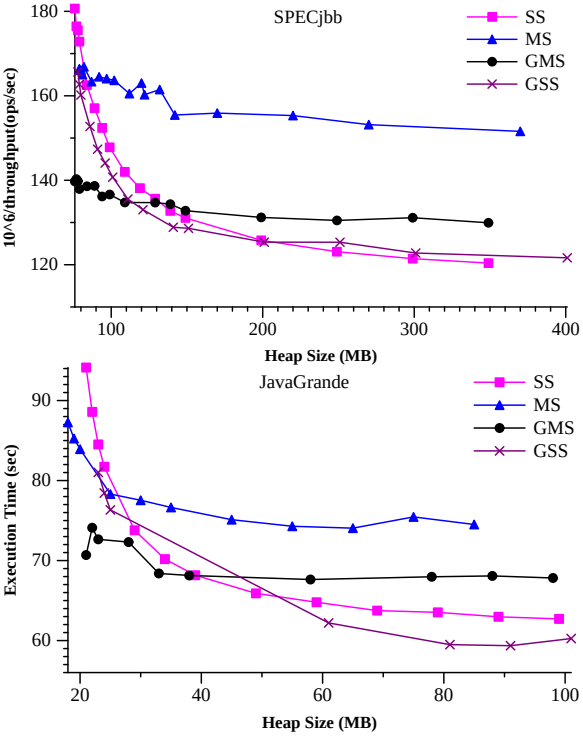

Figure 1: Application performance for different GC systems and heap sizes (lower is better).

by offline, cross-input, as well as online, program and underlying resource characteristics,

- It provides an empirical evaluation of the efficacy of dynamic, application-specific GC selection.

In the following section, we motivate our work and present an overview of our dynamic GC switching framework. Section 3 describes the different approaches we employ to dynamically switch between collectors. Section 4 presents and discusses our experimental results. Section 5 discusses some related work and Section 6 presents our conclusions.

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 [20, 15, 25]. These systems run a single virtual machine (VM) image continuously so that applications and code components can be uploaded and executed as needed by customers (for customization, collaboration, distributed execution, etc.).

Given this model (single VM and continuous execution) and existing JVM technology, a single, general-purpose collector and allocation policy must be used for all applications. However, many researchers have shown that there is no single combination of a collector and an allocator that enables the best performance for all applications on all hardware and given all resource constraints [5, 14, 31]. Figure 1 confirms these findings. The graphs show performance (lower is better) over heap size for the SPECjbb [28] and the JavaGrande [21] benchmarks, executing within the JikesRVM [2].

Figure 1(a) shows that for SPECjbb, the semispace (SS) collector, performs best for all heap sizes larger than 200MB, the generational/semispace hybrid (GSS) performs best for heap sizes 120-200MB, and the generational/mark-sweep hybrid (GMS) performs best for heaps smaller than 120MB. In Figure 1(b), GMS is the best for small sized heaps, SS performs well for heap sizes 40-53MB, and GSS performs best for heap sizes larger than 53MB. We refer to the point at which the best-performing GC system changes as the *switch point*.

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

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

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

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

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

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

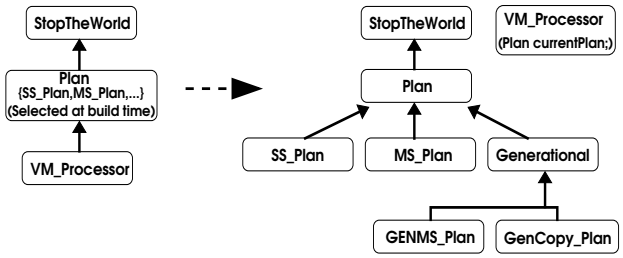

Figure 2: Original and New (Dynamic Switch-Enabled) JikesRVM JMTk Class Hierarchy

marked space is reclaimed.

The GSS system makes use of well-known generational garbage collection techniques [3, 29]. Young objects are allocated in a variable-sized nursery space using bump-pointer allocation. Upon a minor collection, the nursery is collected and the survivors are copied to the mature space. The mature space is collected by performing a semispace copying collection, following a minor collection, as needed. This process is referred to as a major collection.

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

The MS and GMS systems do not use a large object space by default. All GC systems include 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 our switching framework.

2.2 Dynamic GC System Switching

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

Our extension to the JikesRVM requires that multiple GC systems be included in a single system image. To enable this, we implemented a generic *Plan* class, from which all specific GC system classes derive, e.g., SSPlan, MSPlan, GMSPlan, and GSSPlan. Each of these plans are instantiated in a single image of our system. Figure 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 GC system interface to the JikesRVM (*VM_Interface*). This field identifies the GC system that is currently in use. At all times, an object instance of each GC system (a plan instance) is available in our system. The current default allocator that is used, depends on the current collection system. The Plan class invokes a static allocation routine according to the GC system indicated by *currentPlan*. When a switch occurs, the instance of the appropriate collector becomes the global instance of the system.

The immortal space and large object space within our system

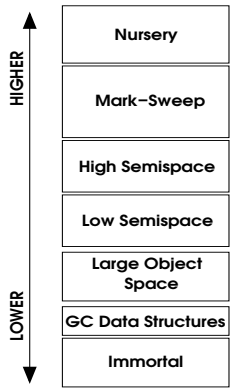

Figure 3: Address Space Layout in the Switching System

are shared across all GC systems. Since the extant versions of MS and GMS do not implement a large object space by default, we extended both to do so. Objects larger than 16KB are allocated from the large object space. To support multiple GC systems, we require address ranges for all possible virtual memory resources to be reserved. We try to make efficient use of the virtual address space and to overlap as many address ranges as possible (Figure 3). Note that these address ranges are mapped to physical memory lazily (as it is needed), in 1 Megabyte chunks.

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

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

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

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

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

Semispace (SS) to Mark-sweep or Generational Mark-sweep.

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

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

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

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

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

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

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

2.3 Implementation Details

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

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

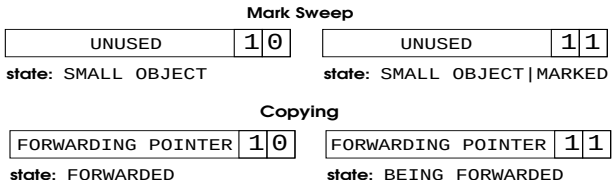

Figure 4: Examples of Bit Positions in the Object's Status Word

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 GC system, the value `0x2` indicates that the object is a small object. However, if instead, the semispace collector is used, this state indicates that the object has been forwarded to the to-space during a collection. Similarly, if the lowest two bits are set, it signifies that the object is a small object and has been marked live by the mark-sweep collector; the same state indicates to a semispace collector thread that the object is currently being forwarded by another thread.

Since we forward marked objects to the semispace, we need to support all four distinct states, along with the space required for the forwarding pointer. To account for the two additional bits required, we use a technique known as bit-stealing [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.

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

A second source of overhead, other than our inability to inline each of the individual allocation routines, is a universal write barrier, which is necessarily introduced by our system. Write barriers are used to record pointers for generational collectors [8, 18] since heap areas are independently collected. Cross-generation (old-to-

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

Table 1: Annotated garbage collection selection decisions for each benchmark

young) pointers must be tracked so that they can be traced during collection of the young (nursery) heap space. Since our system can switch at any time to a generational garbage collector, we must always insert write barriers. However, to make this process as efficient as possible, we use a single, shared write barrier for all GC systems. In our system, the nursery always occupies the highest virtual address range. Hence, we require only a single check to determine if the young object reference is in the nursery. We evaluate the impact of these overheads in Section 4 and discuss solutions for reducing each.

3. GARBAGE COLLECTOR SELECTION

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

Annotation-Guided Selection

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

We annotated programs with GC identifiers for a range of heap sizes. We used a 4-byte annotation in each class file of an application containing a main method. We inserted annotations into class files using an annotation language and a highly compact encoding that we developed in prior work [22]. Upon initiation of dynamic loading of the first application class file, JikesRVM switches to the collection system specified. If an annotated GC system is not available, the default JikesRVM collection system is used (currently, GMS). Since the best-performing collection system may depend on the underlying architecture (memory size, cache 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.

Automatic Switching

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

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

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

4. EVALUATION

To empirically evaluate the efficacy of our framework , we performed a series of experiments using a number of benchmark programs. We gathered results on a dedicated 2.4GHz x86-based single-processor Xeon machine (with hyperthreading enabled) running Debian Linux v2.4.18. We implemented our switching framework within the JikesRVM version 2.2.0 with jlibraries R-2002-11-21-19-57-19. We report performance results using the JikesRVM Fast configuration in which all methods executed are optimized upon initial invocation. We repeatedly executed the benchmarks through a test harness and report the average of the last 5 of 10 runs; therefore, compilation is not included in our data (except for the OptCompiler benchmark which exercises the JikesRVM compilation system). The benchmarks we consider are the SpecJVM98 suite, SPECjbb200, the JavaGrande suite, and the JikesRVM optimizing compiler (first harness run of JikesRVM executing the SpecJVM98 javac benchmark).

4.1 Results

We first present results that show the efficacy of switching using annotation-guided GC selection. The results are shown in Figures 5 and 6 for a number of benchmarks. The x-axis is heap size in megabytes and the y-axis is total time (in milliseconds) for program execution. For SPECjbb, the y-axis is the inverse of the throughput reported by the benchmark; we report this metric to maintain visual consistency with the execution time data, i.e., lower numbers are better. As described in Section 3, we selected the best performing GC system for a range of heap sizes by considering multiple inputs offline: SpecJVM98 – input 10 and 100, SPECjbb – one and two warehouses, JavaGrande – section3/AllSizeA and section3/AllSizeB). We identified the best performing GC system all inputs (see Table 1), for each heap size. For brevity, we present

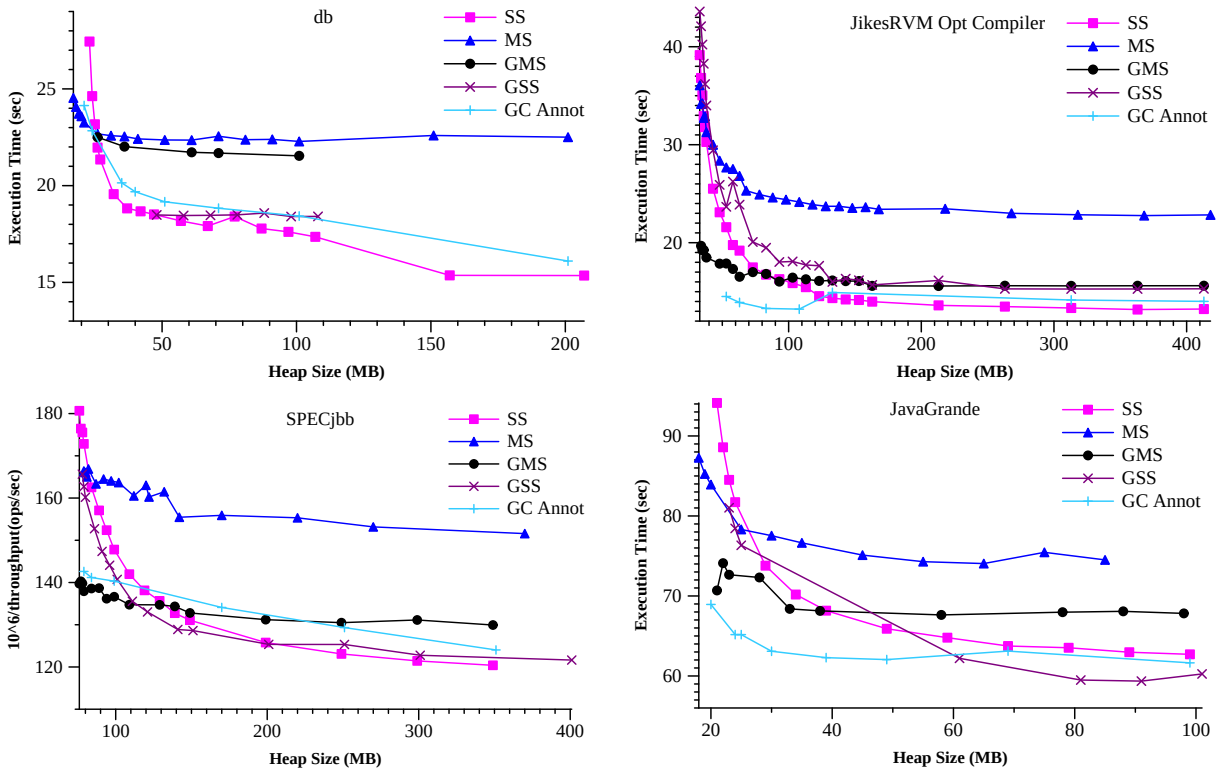

Figure 5: Performance Results. For each benchmark, data for the reference system is shown (SS, MS, GMS, GSS). In addition, the data demarked with (+) shows the efficacy of annotation-guided GC system selection.

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

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

results only for input 100 for SpecJVM98, a single warehouse for SPECjbb, and input section3/AllSizeA for JavaGrande.

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

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

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

The star (*) indicates that the performance of the switching system configurations are not directly comparable since the amount of

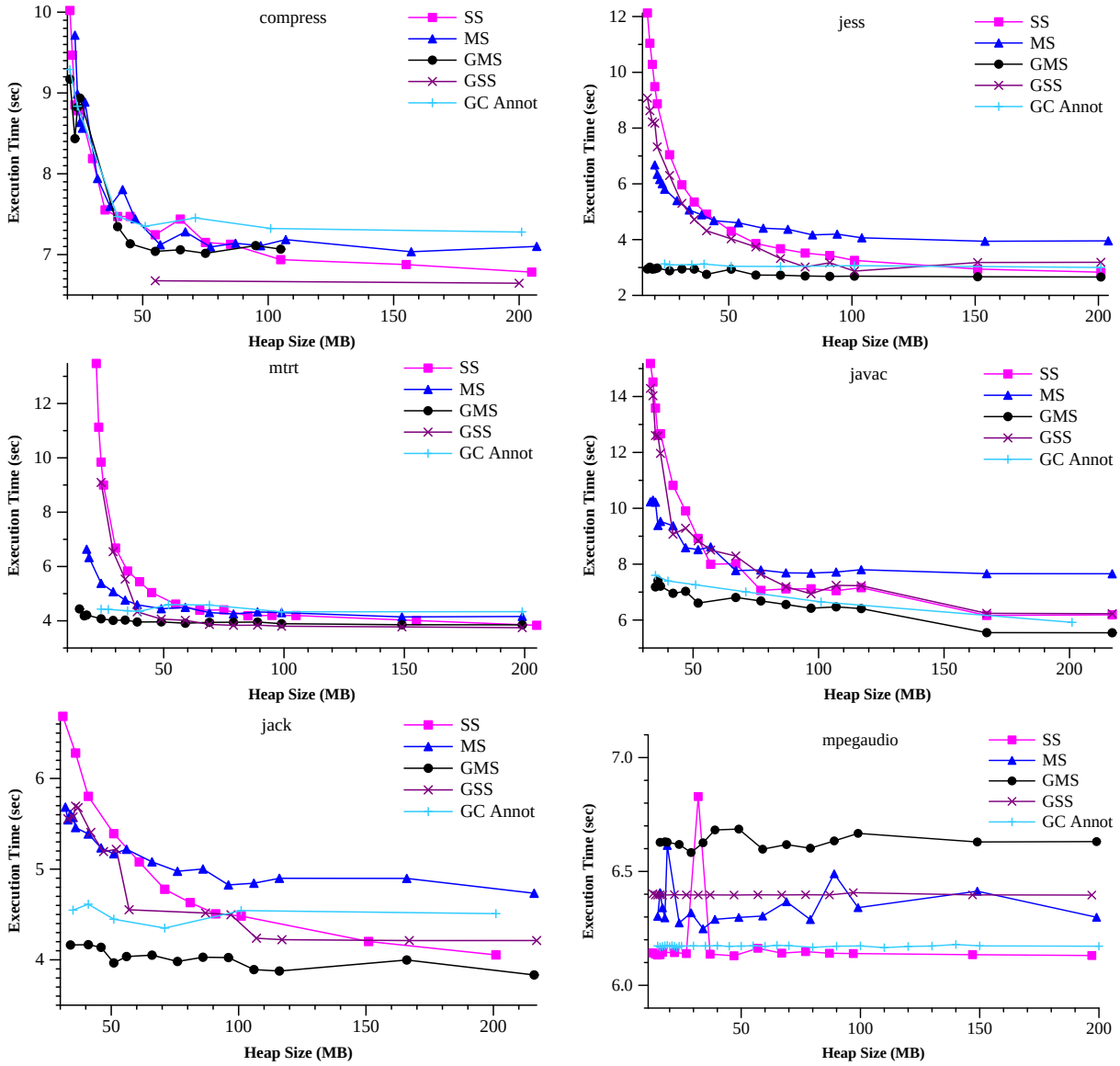

Figure 6: Performance Results, continued. For each benchmark, data for the reference system is shown (SS, MS, GMS, GSS). In addition, the data demarked with (+) shows the efficacy of annotation-guided GC system selection.

code compiled differs: the switching system in column 2 does not inline allocation sites and it inlines write barrier code in column 3. This is only the case for OptComp for which we are analyzing compilation overhead; for all other cases, compilation overhead is not included. The mean values without the OptComp benchmark were 0.89% for column 2 and 0.04% for column 3.

The data indicates that our system does not impose significant overhead. We show execution times for a range of heap sizes for a representative set of benchmarks, in Figure 7 – to point out some anomalies. Again, this data shows the performance of the switching system when no switching is performed; it shows the impact of not inlining allocation sites and having to inline write barriers – for a non-generational semi-space collector.

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

Jack exhibits the worst overall overhead, due to missed inlining opportunities for allocation sites. Inlining of allocation routines seems to always help performance for jack. Javac is also impacted by turning on inlining, however, in a way that is more similar to jess than jack. Inlining of allocation routines with small heaps hurts performance, since the average overhead due to disabled allocation inlining is negative (in Table 2). These results indicate that we should also consider inlining as a parameter to annotation-guided and automatic GC switching.

The mpegaudio performance results are very anomalous and non-intuitive. When we inline allocation sites, performance degrades –

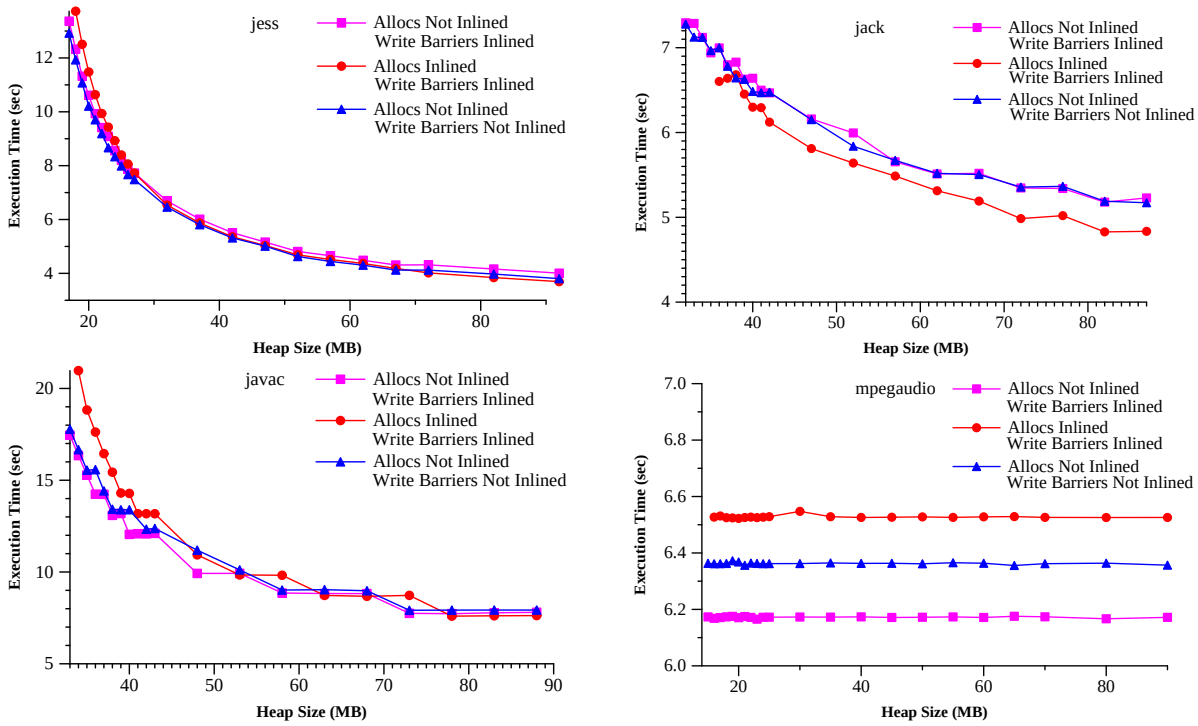

Figure 7: Execution times for some benchmarks with our GC switching system always running the SS collector (squares), allocation routines inlined in the GC switching system (circles), and write barriers not inlined in the GC switching system (triangles), across different heap sizes

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

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

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

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

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

To reduce the overhead of our framework, we can specialize methods according to the currently selected GC system. That is, we can assume that the GC will never change, and as such, inline GC-specific allocation routines. Similarly, we can avoid inlining write-barriers if the current GC does not require them. However, when a switch occurs, we must invalidate the specializations. This requires simple recompilation for most methods. However, methods that are on the runtime stack will require on-stack-replacement (OSR). OSR enables replacing the stack frame of an invalidated method with another (unspecialized version). As part of future work, we plan to build upon existing work to enable both invalidation [17, 4] and on-stack-replacement [13, 17]. The trade-off of performing specialization is recompilation overhead and unoptimized execution time. We plan to empirically evaluate these trade-offs as part of future work.

To summarize our empirical evaluation of annotation-guided GC switching, we next show how our system reduces performance hits taken when the "wrong", i.e., worst-performing collector, is chosen. In addition, we show the average performance degradation over optimal selection. The data in Table 4 shows the average difference between our GC switching system and the best performing system at each heap size (column 2) and between our system and the worst performing system at each heap size (column 3). In

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

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

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

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

parentheses, we show the average absolute difference in milliseconds; for SPECjbb the value in parenthesis is the difference in inverse throughput. Note that the data in this table does not compare our system against a single JikesRVM GC system; instead, we are comparing our system against the best and worst-performing GC system at every heap size. For example, for large heap sizes for the SPECjbb benchmark, the SS system performs best. For small heap sizes, GMS performs best. In this case, to compute percent degradation, we take difference between execution times enabled by our system and the SS system for large heap sizes, and our system and the GMS system for small heap sizes.

In column two (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 across heap sizes, on average, i.e., our system improves performance over the best-performing GC system. This is due to the combined effect of missed inlining of allocation sites and our use of static method invocations that avoid the dynamic dispatch used in the reference system. Our annotation-guided system degrades performance over the best-performing GC system by 4% on average across benchmarks. In addition, it improves performance by 19% over the worst-performing GC system.

Automatic Switching

In addition to annotation-guided GC selection, we also implemented automatic switching using a simple heuristic (based on 60% heap residency). Table 5 shows the average performance difference between our switching framework making use of the automatic decision heuristic (AutoSwitch) and the reference system. The table uses the same format as we used previously for Table 4.

In some cases, e.g., mpegaudio, AutoSwitch performs better than GCAnnot. This is due to differences in the time at which the switching occurs. In both cases, the initial GC is GMS. For GCAnnot, we switch just prior to the start of execution; using AutoSwitch, we switch at the first GC at which heap residency remains as 60% following collection. For mpegaudio, GMS is more appropriate for its initial startup phase, and GC is more appropriate during its steady state. As such, AutoSwitch outperforms GCAnnot. AutoSwitch achieves an average improvement of over 16% over the worst-performing GC system, across all benchmarks, and it imposes overhead that is under 5%. As an initial attempt, our simple heuristic performs quite well for the benchmark programs that we studied.

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 [23, 24], 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. In [26], the authors show that coupling compacting collectors with different performance characteristics can be effective.

Other related work shows empirically that performance enabled by garbage collection is application-dependent. For example, Fitzgerald and Tarditi [14] perform a detailed study comparing the relative performance of applications using several variants of generational and non-generational semispace copying collectors (the variations are due to different write barrier implementations). They show that over a collection of 20 benchmarks, each collector variant sometimes provides the best performance. On the basis of these measurements they argue for profile-directed selection of garbage collectors. However, they do not consider variations in input, require different prebuilt binaries for each collector, and only examine semispace copying collectors. Other studies have identified similar opportunities [5, 31, 27]. IBM's Persistent Reusable JVM [19] attempts to split the heap into multiple parts grouped by their expected lifetimes, employs heap-specific garbage collection 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. However, to our knowledge, no extant research has defined a general, easily extensible framework for switching between very diverse garbage collection systems, such as the one that we describe. In addition, our automatic switching heuristic, albeit simple, requires no user intervention and achieves considerable performance improvement.

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" collection system can be significant. To overcome these limitations,

we have developed a framework that can automatically switch between garbage collection systems without having to restart and possibly rebuild the execution environment, as is required by extant systems. Our system can switch between collection strategies *while* the program is executing. As such, it enables application-specific collection policies to be implemented that can also adapt to the underlying resource availability. The overhead introduced by our system is 4% for both annotation-guided and automatic switching using a simple heuristic, on average. Our system significantly improves performance (19% for annotation-guided and 16% for automatic switching) over the worst-performing collection system, on average.

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] 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] 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).

[12] 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.

[13] FINK, S. J., AND QIAN, F. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)* (San Francisco, California, March 2003).

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

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

[17] HOELZLE, U., CHAMBERS, C., AND UNGAR, D. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)* (San Francisco, California, June 1992).

[18] 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.

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

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

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

[22] 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.

[23] 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.

[24] 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).

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

[26] 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.

[27] 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.

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

[29] 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).

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

[31] 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.