Program-level Adaptive Memory Management

Chengliang Zhang†, Kirk Kelsey†, Xipeng Shen†,
Chen Ding†, Matthew Hertz∗, and Mitsunori Ogihara†

†Computer Science Department
University of Rochester
{zhangchl,kelsey,xshen,cding,ogihara}@cs.rochester.edu

∗Computer Science Department
Canisius College
hertzm@canisius.edu

Abstract

Most application's performance is impacted by the amount of available memory. In a traditional application, which has a fixed working set size, increasing memory has a beneficial effect up until the application's working set is met. In the presence of garbage collection this relationship becomes more complex. While increasing the size of the program's heap reduces the frequency of collections, collecting a heap with memory paged to the backing store is very expensive. We first demonstrate the presence of an optimal heap size for a number of applications running on a machine with a specific configuration. We then introduce a scheme which adaptively finds this good heap size. In this scheme, we track the memory usage and number of page faults at a program's phase boundaries. Using this information, the system selects the soft heap size. By adapting itself dynamically, our scheme is independent of the underlying main memory size, code optimizations, and garbage collection algorithm. We present several experiments on real applications to show the effectiveness of our approach. Our results show that program-level heap control provides up to a factor of 7.8 overall speedup versus using the best possible fixed heap size controlled by the virtual machine on identical garbage collectors.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection), Optimization

General Terms Algorithms, Languages, Performance

Keywords garbage collection, paging, adaptive, program-level, heap sizing

1. Introduction

Their is a strong correlation between memory allocation and program performance. Traditionally, this relationship has been defined using the concept of a *working set*. An application's working set is the set of objects with which it is currently operating. The amount of memory needed to store these objects is called the application's *working set size*. When the available memory is less than an application's working set size, throughput is limited by the time the application spends waiting for memory to be paged in or out. Until the working set size it reached, we can improve program performance by increasing the available memory and thereby reducing the number of page faults. Once an application's working set fits into main memory the application stops paging, so increasing memory further may not improve performance.

Garbage collection automatically reclaims memory allocated dynamically and thus relieves the programmer of the need to explicitly free blocks of memory. Increasing the memory available to a garbage-collected application also tends to decrease its execution time, but for a different reason. By increasing the size of its heap, an application can perform fewer garbage collections and thereby improve its throughput. While we could size the heap to just fit the working set, this would require the collector to spend more time collecting the heap and increase execution times accordingly. Setting a larger heap size in the virtual machine reduces the need to collect the heap and, generally, reduces the total time spent in collection. Selecting larger heap sizes, therefore, improves performance by reducing the overhead due to garbage collection.

While decreasing the frequency of garbage collection, using larger heaps may not improve performance. First, as the heap grows, individual collection may take longer as the collector examines more objects. These larger heaps also increase the changes the heap will not fit into memory and must have pages evicted to virtual memory. Another downside is that the live data in the heap may be scattered over a larger area. This scattering reduces data locality and hurts the performance of caches and the TLB. Ultimately, larger heap sizes requires balancing the benefit of fewer collections with the costs of reduced locality and paging.

The exact trade-off between frequent GC and paging is hard to predict, because it really depends on the application, the virtual machine, the operating system, other programs in the environment, and all of their interactions. We will show that the relationship follows a common pattern. This pattern includes an optimal heap size value, which we can adaptively identify during the program execution to minimize the execution time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISMM'06 June 10–11, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-221-6/06/0006. . . $5.00.

	Scope of adaptation	Changes Required
PAMM	VM, OS, Program	Program
Automatic Heap Sizing [22]	VM / OS	VM / OS
GC Hints [5]	Program	Program
BC [11]	VM /OS	VM / OS
Preventive GC [9]	Program	Program

Table 1. Comparison of different adaptive memory management schemes

There are two levels at which the heap size can be controlled: at the program level and the virtual machine level. Past work has examined this problem in a number of ways. Yang et al. estimated the amount of available memory using an approximate reuse distance histogram [22]. A linear model controlled adjustments to the heap size so that the physical memory was utilized fully. Maintaining these histograms required modifying the operating system and so their evaluation was done in simulation. An alternative approach by Hertz et al. modified the garbage collector and OS to avoid touching pages written to the backing store and thereby reduced the cost of collecting a large heap [11]. Andreasson et al. used reinforcement learning to find the best times to perform garbage collection. Because it must first be trained, this strategy is effective only after tens of thousands of time steps (decision points) have passed [2]. We note that each of these schemes made changes to the operating system and/or virtual machine.

Programs usually run in repetitive patterns. We call each instance of the pattern a *phase*. While different compilers (or virtual machines using just-in-time compilation) produce different optimized code, the compilers do not alter the relationships between phases. As a result, a program expresses surprisingly uniform behavior across phases which can be used to adapt the heap size. Even better, this adaption can be independent of the compiler and virtual machine, since the phase behavior is also independent. Moreover, this approach is independent of the underlying architecture and memory management scheme.

Based on the program phase information, we propose a method of program-level adaptive memory management, or *PAMM*. We add the PAMM controller using program instrumentation. The controller acquires data available from many layers of the system, such as the operating system, virtual machine, and the application itself. By using data such as the current size of the heap and the number of page faults incurred, our controller can calls for GC when it is needed. Table 1 compares PAMM with several previously proposed adaptive memory management schemes.

The rest of the paper is organized as follows: Section 2 provides the background information of phases. Section 3 describes the program level memory management scheme in detail. Section 4 presents the implementation details and the benchmarks we used. Section 5 presents our experimental results on the six benchmark applications and compares them with the brute force search method. Section 7 discusses the related word and Section 6 concludes our work.

2. Behavior Phases in Programs

Many programs have repetitive (yet input-dependent) phases that users understand well at an abstract, intuitive level, even if they have never seen the source code. For example, a database iterates over the data records to find the results matching a given query and a server application processes incoming requests one by one. Other

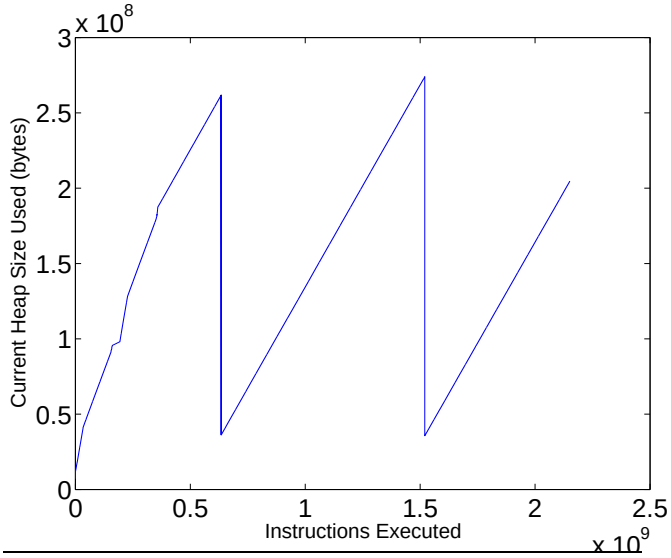

Figure 1. Phases detected for pseudoJBB. The heap size increases steadily and evenly until a GC is called.

programs like programming environment tools, compression and transcoding filters, and interpreters have similar phase behavior. While a phase usually spans many functions and loops, it has only one starting point and one ending point and may occur multiple times during the program run.

Program-level adaptive memory management is based upon phase information for three important reasons. First, phases provide a high level summarization of a program. The behavior of the phase instances are quite similar, so we do not need to measure every possible phase to get a reasonable measurement. Second, phases are very repetitive, so the memory usage of the program is split evenly by the phase boundaries. Third, a phase usually represents a memory usage cycle. Garbage collection is best performed at a phase boundary, because it is at these points in the program that all temporary objects will be dead and ready for collection.

One way of detecting the phases is through active profiling. Active profiling uses regular inputs to induce behavior repeatable enough for analysis and phase marking.Other techniques have also been proposed previously and could be used for program-level adaptive memory management. For instance, Georges et al. selected Java methods whose behavior variation is relatively small [10]. There are also many algorithms which exploit aspects of the program structure such as loops and procedures [4, 10, 13, 16], regions [12], and call sites [15] to determine phase boundaries.

Figure 1 shows the phase information for a run of pseudoJBB with a 512MB heap. The x-axis of this figure is the logical time within a run and the y-axis shows the heap size at a point in time. The two drops in the figure correspond with the two GCs calls made. We can see that the phases split the memory usage evenly. Figure 2 provides us more detailed information. Each vertical dotted line shows the boundary of 100 phase instances that consume fewer than 150K bytes. Because of the way in which the virtual machine reports the heap size, we cannot obtain more detailed memory usage statistics. As objects are allocated, the heap size does not always increase because new objects can be placed in the fragmented space between existing objects. As a result, we see multiple phase instances with the same heap size.

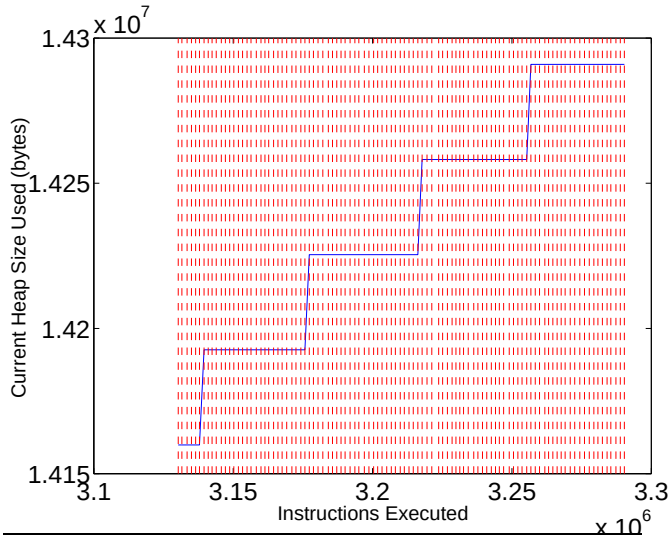

Figure 2. A detailed view of the phases of pseudoJBB. Each vertical dotted line corresponds to a phase and there are 100 phases in this figure. Because of limitations of the VM, we cannot get more detailed heap size information.

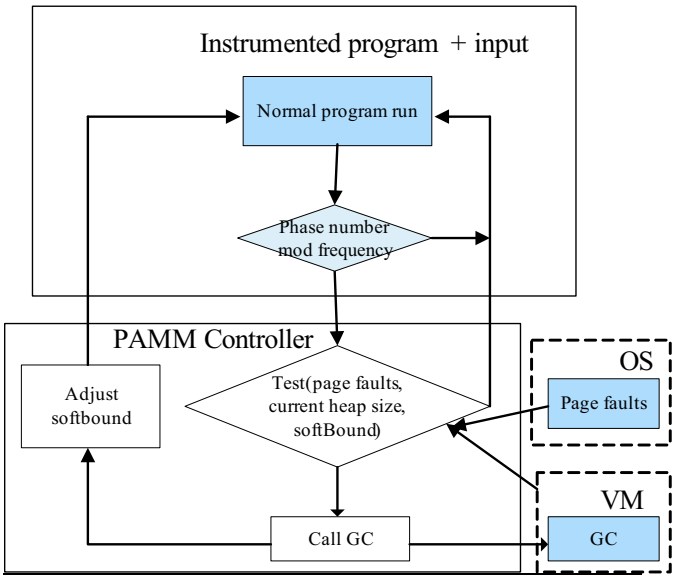

Figure 3. Flow graph of program-level memory management

3. Program-level Memory Management

For ease of this discussion, we begin by introducing our terminology. The *VM controlled heap size* is the target heap size, often specified via a command-line argument, that is maintained by the Virtual Machine. We also call this size the *VM bound* or *hardbound*. By contrast, the *program controlled heap size* is the target heap size maintained at the program level. We also call the program controlled heap size the *program bound* or *softbound*. Note that the program cannot control the space that GC needs, although the VM can. The size of the heap at any instant in the program run we call the *current heap size*. The current heap size can be computed by subtracting the free heap memory from the total memory.

Figure 3 shows the control flow graph of our program-level adaptive memory management approach. This begins with us instrumenting the original program to get the phase information. Then, during the program run, we invoke the test function each time we finish executing a specific number of phase instances. ¿From the current heap size and the number of page faults the process has incurred, the controller decides whether or not to make a GC call. After the GC completes, the PAMM controller adjusts the softbound using the number of page faults duing the garbage collection and the reason the heap was collected.

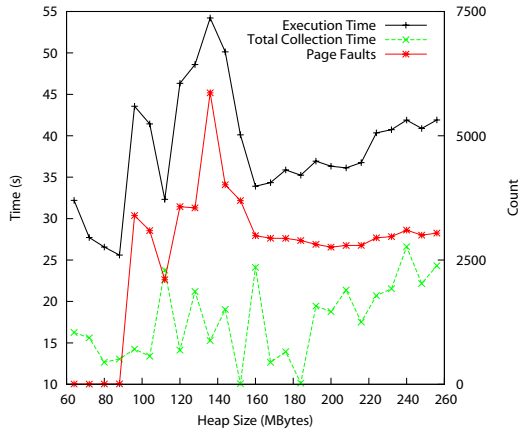

Figure 4. Behavior of pseudoJBB using GenCopy with 192M physical memory

3.1 Monitoring Frequency

There are two principles used to decide upon the monitoring frequency. The monitoring must be frequent enough to provide timely information when deciding to invoke the collector. But it must also be infrequent enough that it does not impose a substantial overhead. On average, each call to the test function takes about 0.14 milliseconds. Thus, for most applications, we can safely perform thousands of checks without dragging down program throughput.

Phases boundaries provide us with many potential opportunities to check whether the heap must be collected and, when needed, invoking the collector. We cannot just check at every phase boundary, however. For example, pseudoJBB has as many as 221,804 phases and, were we to call the test function at each of these, we would add more than 30 seconds to the running time of this minute-long program. In practice, we avoid this problem by specifying a frequency for checks (e.g., n pseudoJBB we check only once every 100 phase boundaries). By calling the test function less frequently an execution performs only slightly more than 2000 checks, which is sufficient for deciding upon tens of GC calls.

Manual selection of a suitable frequency is easy. Dynamic selection is also possible according to the two principles in the above. The adaptive solution would be useful if the program is composed of several different types of phases and the frequency can not be fixed. In our experiments, we use the manual solution.

3.2 Making Decisions

Figures 4 through 6 show the behavior of pseudoJBB running at different hardbounds using 3 different GC algorithms: GenCopy, CopyMS, and MarkSweep. In all these figures, the page faults include both those caused by the mutator and those caused by the garbage collector. The total collection time includes delays in collection that occur as a result of paging.

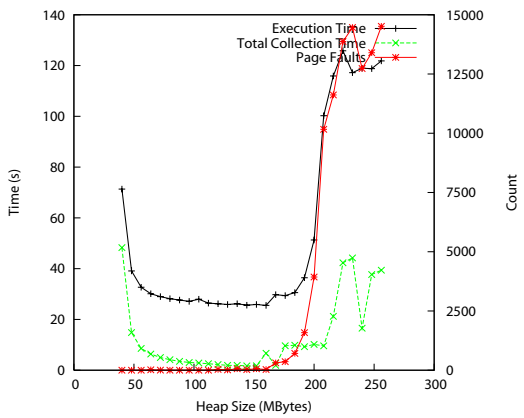

Figure 5. Behavior of pseudoJBB using CopyMS with 192M physical memory

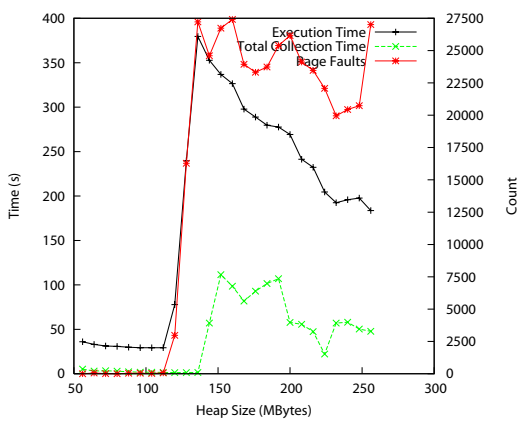

Figure 6. Behavior of pseudoJBB using MarkSweep with 192M physical memory

We can observe that for each GC scheme, the total execution time first drops down, then quickly goes up and drops down slowly at last. There exist multiple locally optimal hard bounds. However, there does exist an optimal point, such that:

1. When the hard bound is smaller than the point, the total collection time correlates with the total execution time very well and the number of pages faults remains steadily low.
2. When the heap size goes beyond the point, both the total execution time and the number of page faults increase dramatically. However, the number of pages faults correlates better with the execution time.

Based on these observations, we propose our program-level adaptive control scheme. We set a large VM bound for the virtual machine when we start the virtual machine. We also maintain a softbound. At each monitoring point, we poll the virtual machine for the current heap size of the program and poll the OS the number of page faults accumulated from the last GC call. For one case, if the current heap size is greater than the softbound, we instruct Jikes RVM to perform a garbage collection. By this, we control the heap size from the program level. We call this case *a reach-softbound invocation*. For the other case, if that number of page faults since last GC is above a certain level, we consider that the softbound goes beyond the optimal point and invoke GC. Considering that the system may be not stable, for the second case, we add the requirement that the current heap size should be bigger than the heap size after last GC call by at least 2M. WE call this case *a paging invocation*.

The number of page faults caused by the current process can be obtained from the file "/proc/self/stat". This file is actually a pseudo file and is used as an interface to the kernel data structure. Since it is maintained in the memory, it is efficient to read from it.

3.3 Adjusting the Softbound

After the GC call, our controller need to change the softbound to adapt to the environment. These includes two possible actions: increasing or decreasing.

To help adjusting the softbound, we introduce two new variables: left mark and right mark. The left mark is the very softbound where we increase our softbound and the right mark is the very softbound where we decrease our softbound. We also assume there is a predefined step size. In our experiment, the step is set to be 10M.

Our initial softbound is set at the right beginning of the program. It is currently set to be the current heap size plus 2 steps. The left mark is set to be the current heap size.

The softbound is increased as follows: First, the current softbound is recorded as the left mark. If right mark does not exist, the softbound is increased by the step size . Otherwise, it is to be the mean of the current softbound and the right mark. However, there do exist cases where the right mark is incorrectly set to be smaller than the optimal point because of the changes in the environment or some random reasons. To get over those cases, we force both the softbound and the right mark to increase by 1M, if their distance is smaller than 2M.

The softbound is decreased as follows: First, the right mark is set to be the current softbound. We then move the softbound to the mean of softbound and left mark. To deal with the case in which the left mark is incorrectly set, we force the softbound and the left mark to decrease by 1M if their distance is smaller than 2M. Since it is of no use if the soft bound is smaller than the current heap size after GC, we make sure that the smallest value of softbound is 1M away from the current heap size.

From the previous description, it is easy to see that we actually follow a binary search scheme to find the optimal softbound.

The next question is when to invoke increase and decrease. We measure the page faults incurred by the current GC call. If the number is smaller than a specified threshold, e.g., 10 in our experiment, and the GC is a reach-softbound invocation, we consider there is still room to boost the performance, so we increase the softbound. In the other case, if the number of page faults is no smaller than the threshold, we decide to decrease the softbound, regardless the reason the type of GC being triggered.

4. Experimental Methodology

To evaluate the effectiveness of adaptive heap sizing we must compare the total execution time for a number of benchmarks. For this analysis we compared the results on 4 SPECjvm98 benchmarks [8], as well as pseudoJBB [7] and ipsixql [17]. *201_compress* is a high-performance application to compress and uncompress large files, based on the Lempel-Ziv method. In our experiment, it compresses and uncompresses 5 different files 5 times each. We have identified 2 different phases, one is within the compress process and the other is in the decompress process. *202_jess* is a Java expert

Benchmark	Phasemarks	Monitoring frequency	Parameters
201_compress	2	32768	-s100 -M1 -m1 -a
202_jess	2	100	-s100 -M1 -m1 -a
209_db	1	10	-s100 -M1 -m1 -a
227_mtrt	1	100	-s100 -M1 -m1 -a
ipsixql	8	100	1, 7
pseudoJBB	8	250	140000

Table 2. Benchmarks and their parameters

shell system based on NASA's CLIPS expert shell system. In our experiment, it is used to solve the Number Puzzle Problem. We have identified two different phases: one is to parse the expressions and the other is to execute a simple function call. *209_db* is a small database management program that performs several database functions on a memory-resident database. The only phase we identified is to process one data function. *22t_mtrt* is originally dual-threaded ray tracer. However, since we currently cannot manage multiple-threaded programs, we changed it to be single-threaded. The only phase identified in the benchmark is the rendering of a pair of pixels. *Ipsixql* is an XML database program from the DeCapo benchmark suite with a set of 7 queries. Because we identified the phases of Ipsixql via instrumentation with Soot [21], we do not know the meaning of the 8 phases. *pseudoJBB* is a single threaded simulation of a warehouse system which repeatedly process 6 different types of transactions. It is modified from SPECjbb benchmark to perform only a fix number of random loads, and has 7 phases identified during the warehouse initialization. The other phases process transactions one by one. Table 2 shows the related information for each benchmark we used.

All of the experimentation is done using the Jikes compiler (version 1.22) and Jikes RVM (version 2.4.0). When the program-level adaptive memory management is active, the virtual machine is instructed to use a 512 Mbyte heap. In order to have a realistic evaluation of running time, we need to execute the benchmark suite with any optimizations that would typically exist. At the same time, we wish to separate the program execution time from the virtual machine's optimizations. We use the second-run technology first proposed by Bacon et al. [3]. Prior to the timed executions we run the benchmark once with adaptive optimization active. This allows the virtual machine to recompile hot methods. During the timed executions the adaptive optimization system is deactivated so that each trial will be the same. This allows us to isolate the program running time from the optimization overhead. The execution times reported are the minimum of three trials.

The Jikes RVM can be built with several different garbage collectors and optimization schemes. In addition to testing the various benchmarks under different collection routines, each benchmark is evaluated using two different optimization schemes. In the FastAdaptive builds the included classes have all been compiled with the optimizing compiler, and the adaptive compilation of hot methods is done [1]. In the BaseBase builds, the optimizing compiler is not used, and adaptive compilation is unavailable.

All of the benchmarks were evaluated on 2GHz Pentium 4 processors running Linux kernel version 2.6.12 with 512Mb of physical memory. To simulate a more constrained system, we limit the amount of system memory available to the virtual machine. This limited-memory effect is achieved by pinning memory pages in the operating system. For each benchmark the physical memory is artificially constrained to values between 96 Megabytes and 192 Megabytes.

We use two different approaches to identifying the phases of the benchmarks, depending on whether or not the source code is available. In the simpler case where the source is available, the phases are identified manually. Because the function of the benchmark is known, we can insert control at points that are known to execute once per phase. When the source code is not available for direct modification, we use the Soot Java optimization framework to identify the insertion points through profiling. Using Soot, we analyze the number of times a particular instruction in the Java code is executed. Based on the statistics information, we determine the phases manually, and insert the control mechanism there.

4.1 Garbage Collectors

Because we want the adaptive heap sizing to be dependent only on the program behavior, we will illustrate its advantages in the presence of three different garbage collectors: Mark-Sweep, CopyMS, and GenCopy. These collectors are hybrids of other collection schemes. All of them have in common that they are "stop the world" approaches that halt the program mutator during collection.

Basic **Mark-Sweep** (MS) garbage collection traverses the entire object reachability graph. Each object is marked when it is scanned during the search, and unmarked objects are known to be garbage. In the Jikes RVM objects are allocated in blocks of specific sizes, which are maintained in a free-list. Objects that are not marked are simply returned to the list.

Copying memory management allocates objects sequentially into one space of memory. When the space becomes full, the objects are traced as in Mark-Sweep. The difference is that rather than marking objects and managing the free space at the end, objects are copied into another memory space when they are touched. The end result is that the second space has reachable objects consolidated at the beginning, and new objects can again be allocated sequentially after them. The scheme toggles back and forth between two memory spaces.

Generational garbage collection also uses multiple spaces of memory. New objects are allocated into the "nursery," and copied into a "mature" space. The major departure from the copying scheme is that the spaces are not swapped. New objects are always allocated into the nursery, and collection is only performed on regions that are full. This allows collection to be concentrated on the newer objects, which are more likely to be garbage. A write barrier must be used to track references from older spaces into younger spaces to avoid scanning them for reachability.

CopyMS uses two memory regions. New objects are allocated sequentially into the first region, which is a copying space. When the region is filled, reachable objects are copied into the second space, which is managed using Mark-Sweep. No write barrier is present, and every collection is performed over the whole heap.

GenCopy is a generational scheme in which a mature space is managed with a standard Copy approach . In the Jikes implementation, the nursery size is unbounded, so initially the nursery fills the entire heap. Each time the nursery is collected its size is reduced by the size of the survivors. Whole heap collection is done only is the nursery size falls below a static threshold.

The memory requirements and usage of the garbage collection schemes are very different. In a Copying collector the virtual machine must allocate an area of memory twice as large as the heap size being used. Additionally, it will potentially touch every page in the page working set. In the Generational approach, the amount of memory needed does not need to exceed the heap size, as space is transferred from the nursery to the mature space when objects are moved. In most cases, the collector will only have to touch memory pertaining to the youngest elements, which are likely to be in mem-

[1] "Fast" actually indicates that a fully optimized compilation is done, but the assertions are removed.

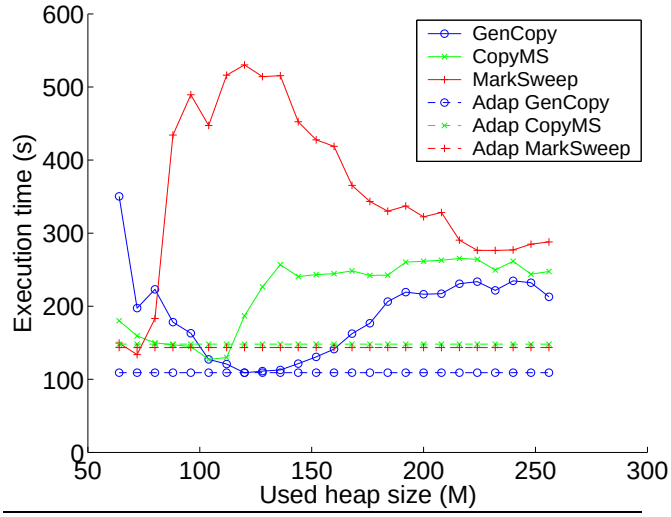

Figure 7. Execution time of pseudoJBB vs. heap size with 128MB physical memory, BaseBase GC scheme is used

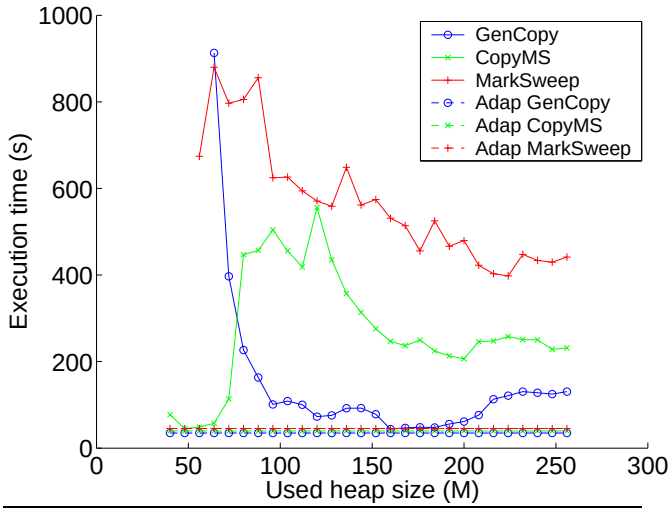

Figure 8. Execution time of pseudoJBB vs. heap size with 128MB physical memory, FastAdaptive GC scheme is used

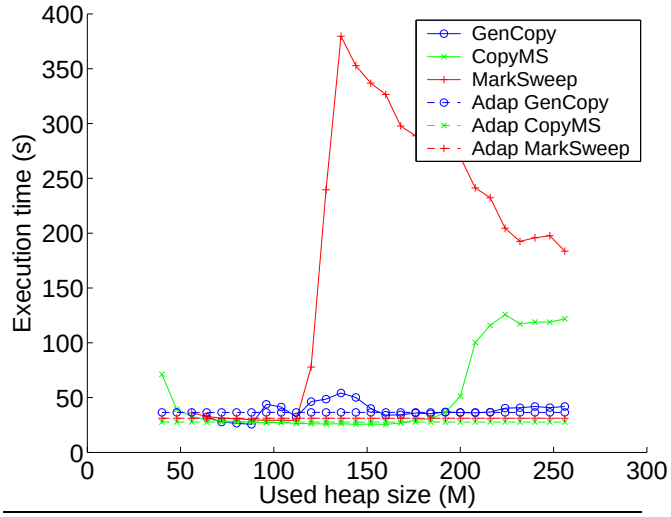

Figure 9. Execution time of pseudoJBB vs. heap size with 192MB physical memory, FastAdaptive GC scheme is used

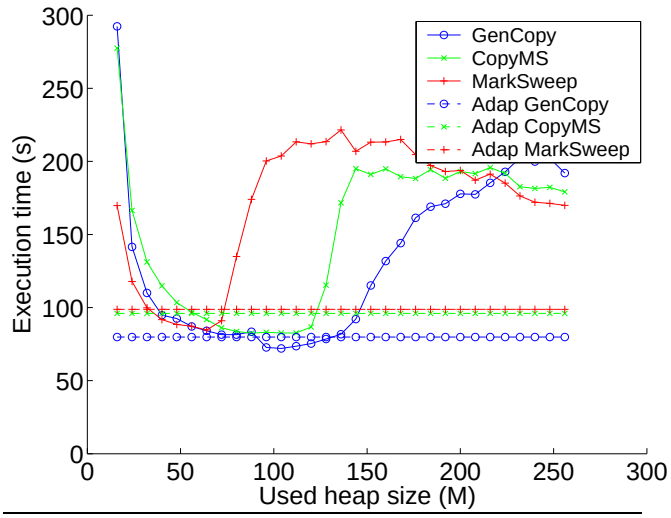

Figure 10. Execution time of ipsixql vs. heap size with 128MB physical memory, BaseBase GC scheme is used

ory. In the Mark-Sweep case, all of the objects may be touched, but no extra memory needs to be explicitly reserved[2].

5. Experimental Results

The curve we wish to optimize can be seen in Figure 7, where the total execution time can be seen to drop to a minimum, rise again as paging becomes more pronounced, and then slowly drop as the frequency of garbage collections becomes trivial. The general shape of this curve is the same for each of the garbage collectors, and is constant for the adaptive cases that each attempt to identify an optimal heap size for the given main memory size. Because the adaptive approach responds to changes in program behavior, the optimal heap size will likely not be found immediately. The result is that the total execution time with adaptive heap sizing may not be as low as the optimal execution time.

[2] Clearly, some internal fragmentation will occur using free-lists of blocks.

Figure 16 depicts the exact heap size immediately before each invocation of the garbage collector during an execution of pseudoJBB using the program-level adaptive memory management. This figure is meant to illustrate how our strategy continually adapts the heap size for pseudoJBB.

We can see that the curves of MarkSweep and CopyMS collectors quickly converge to a stable point in as few as 4 phases. If we look at the detailed type of each GC, we will find that most of the first few are *reach-softbound* invocations and the late ones are mostly paging invocations. Actually, for all of the examples, the page faults play an important role in finding the optimal softbound. After the softbound is stabilized, the heap size before GC still increases at a slow rate, however, the GC now incurs very few page faults. A reasonable explanation is that due to the paging, the process pushes those useless pages out.

For GenCopy, all of the collections are performed on the nursery space. The curve of GenCopy keeps increasing and does not stabilize until phase 99,892. Prior to that there are two places where

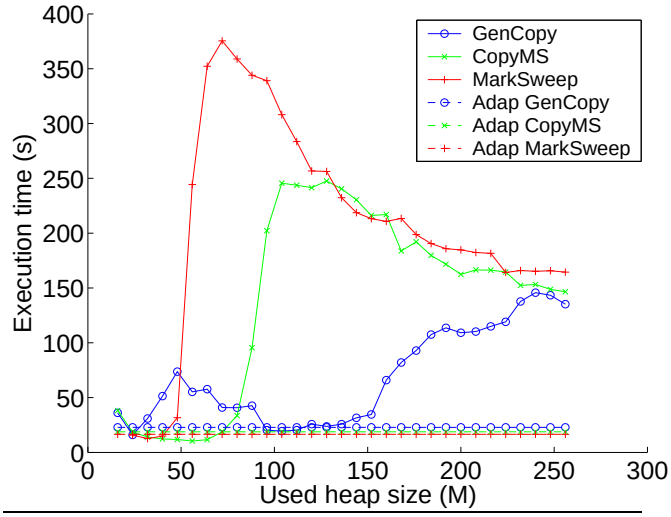

Figure 11. Execution time of ipsixql vs. heap size with 128MB physical memory, FastAdaptive GC scheme is used

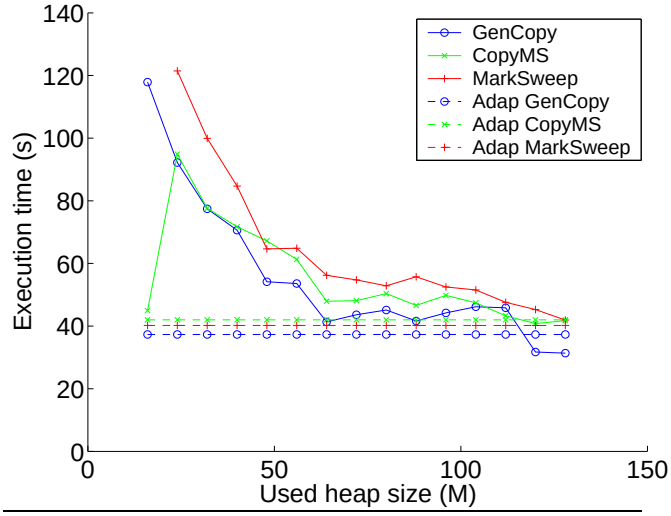

Figure 12. Execution time of 201_compress vs. heap size with 96MB physical memory, FastAdaptive GC scheme is used

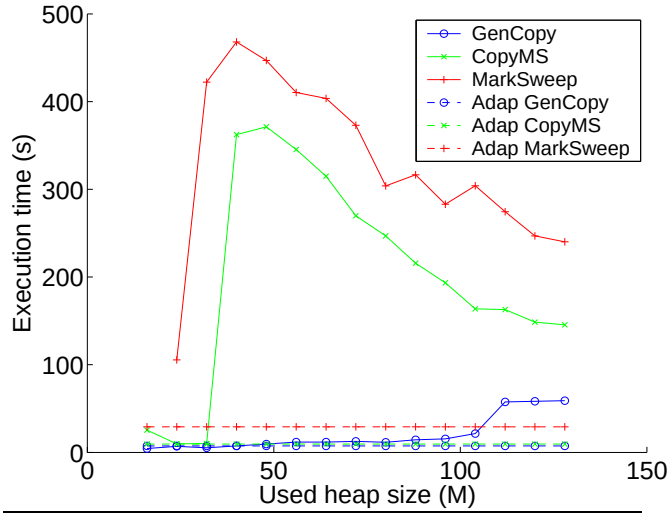

Figure 13. Execution time of 202_jess vs. heap size with 96MB physical memory, FastAdaptive GC scheme is used

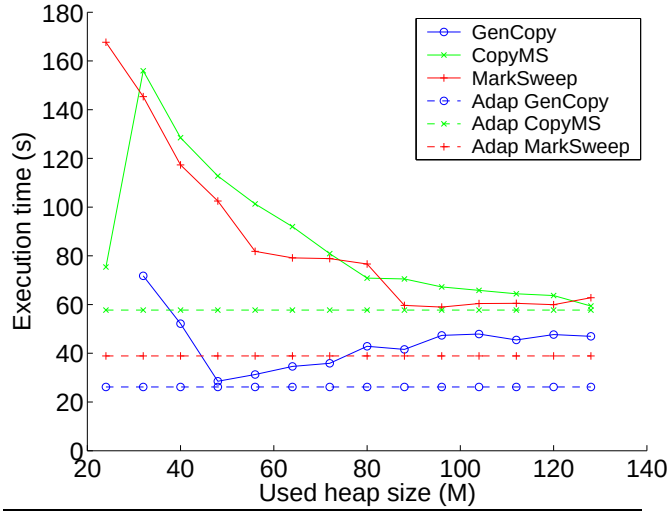

Figure 14. Execution time of 209_db vs. heap size with 96MB physical memory, FastAdaptive GC scheme is used

a lot of collections are initiated. These two places coincide with the warehouse initializations. The initializations incur a lot of page faults and cause the GC to be called frequently. After the initializations, the system quickly stabilizes. At last, though the heap size still increases, each garbage collection takes only tens of milliseconds and causes very few page faults.

In the case of ipsixql with 128 megabytes of physical memory using the FastAdaptive MarkSweep build of the Jikes RVM, the running time at the optimal heap size was 12.6 seconds, while the running time was 207.7 seconds on average and 375.4 seconds in the worst case[3]. The optimal heap size results in an execution time that is 16.46 times faster than the average case. Averaged over all of the benchmarks we have evaluated, the mean execution time for a benchmark is 4.9 times longer than the minimum execution for the same benchmark.

[3] The worst case here is only considering those executions that complete. If the heap size is extremely low, the execution may fail entirely.

While it is clear that finding the optimal heap size results in significant performance gains, it is also the case the finding the optimal heap size is not trivial. The optimal value depends on the garbage collector, the amount of physical memory present, and the particular program being executed, and the compiler optimizations that have been used.

Looking at Figure 7 we see that there is a significant difference between the performance of the MarkSweep garbage collector and that of the CopyMS collector. Not only are the execution times quite different, but their behavior is such that one improves while the other worsens. While the choice of garbage collector can have a significant impact on program performance, the results shown in Figure 8 show that with program-level adaptation the specific garbage collector may become inconsequential. While this is not the case for every benchmark (particularly when the available memory is quite constrained), it is the common case.

The optimal heap size is also affected when the compiler optimizations are changed. Figures 7 and 8 illustrate that the opti-

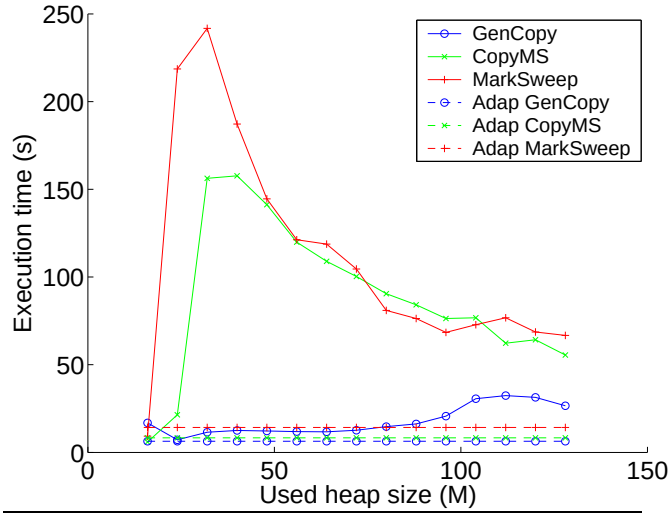

Figure 15. Execution time of 227_mtrt vs. heap size with 96MB physical memory, FastAdaptive GC scheme is used

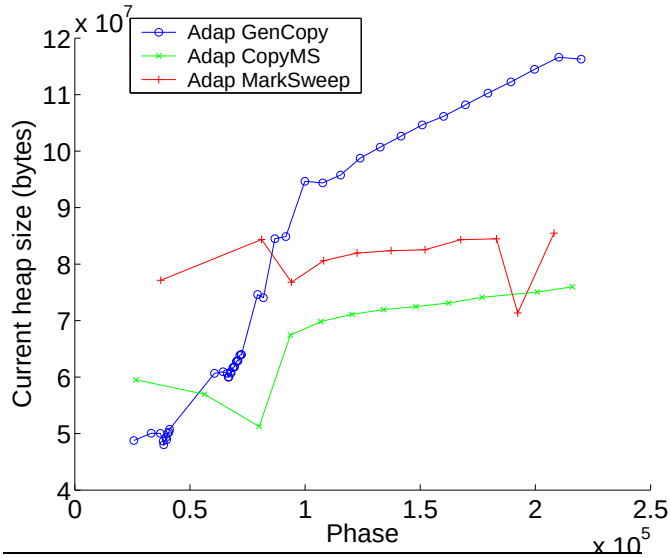

Figure 16. Heap size before the GC, tested for pseudoJBB using 192M physical memory, all GC schemes use FastAdaptive

mal heap size for GenCopy (the best collector for that benchmark) changes from 120 megabytes to 160 megabytes. By comparing Figure 8 to Figure 9 we can identify the impact of changing the amount of physical memory on the execution times. In this case the optimal heap size for the GenCopy collector changes again from 160 to 88 megabytes.

The results depicted in Figures 8 & 11 support the intuitive notion that that the optimal heap size also depends on the executed program. Here the CopyMS collector has the optimal performance in both cases, but the optimal heap size changes from 48 to 56 megabytes. Because of the number of factors that come together in determining the optimal heap size, identifying it prior to execution is impossible.

Table 3 lists the speedup of the adaptive scheme over the best, the second best, and the third best performance of VM-controlled heap sizes for the three garbage collectors. On average across all programs, memory and compiler configurations, the adaptive scheme is 9% slower than the best performance of GenCopy, 10% slower than CopyMS, but 113% faster than MarkSweep (due mainly to the large improvements in two cases). For GenCopy, the largest slowdown is 39% for jess, and the greatest improvement is 26% for optimized pseudoJBB on 128MB memory. For the other two garbage collectors, the speedup ranges from -45% to 780% compared to the best performance of VM-controlled heap size. The slowdown is due partly to the overhead of the run-time controller and partly to the startup cost and mis-steps in the adaptation. On the other hand, since the adaptive scheme uses different heap sizes for different stages of the execution, it achieves a faster speed than the best fixed heap size for over a third of test configurations. In the extreme case, optimized pseudoJBB on 128MB memory using MarkSweep, the adaptive heap control is 7.8 times faster than the best fixed heap size, even though both schemes use an identical garbage collector.

In Table 3, we also compare our adaptive scheme with the default Jikes RVM setting. The default setting of Jikes RVM has initial and maximum heap size to be 50 megabytes and 100 megabytes for FastAdaptive case, and 20 megabytes and 100 megabytes for the BaseBase case. We can see that on average across all of the benchmarks, our strategy have 64% speedup for GenCopy, 524% speedup for CopyMS and 553% speedup for MarkSweep. Our adaptive scheme works very well for all of the benchmarks except pseudoJBB. We are uniformly slower than the best fixed heap size for pseudoJBB benchmark running with 192 megabytes physical memory. A reasonable explanation is that 192M physical memory is big enough for the default GC without too much page faults.

It is interesting to consider the situation that a person needs to select a uniform heap size for all benchmarks with physical memory fixed. We consider pseudoJBB and ipsixql when physical memory is 128 megabytes. For GenCopy, CopyMS and MarkSweep, people will select 128M, 80M and 224M separately. Our adaptive scheme outperforms these selected heap size by 1.6%, 6.5% and 9.4% separately. When the physical memory is 96 megabytes, people will select 48M, 24M and 128M separately for the SPECjvm98 benchmarks. Our adaptive scheme is faster than these selected heap size by 1.4% 1.8% and 3.9% separately.

6. Conclusion

In the presence of automatic memory management, the relationship between allocated memory and application performance becomes more complicated. Allowing a larger heap size will reduce the frequency of garbage collections. Once the heap size exceeds the available physical memory, portions of the heap will be paged to the backing store. Paging is particularly detrimental when garbage collections are performed because the collection is likely to access every page of the heap, thus incurring additional paging overhead. Somewhere on the continuum of heap sizes lies a balance between frequency and cost of collection.

We have introduced a scheme for adaptively identifying the optimal heap size for a program while it is running within a Java virtual machine. We are able to use phase level behavioral information to monitor a program's performance. By observing how the execution time responds to changes in the heap size we can force garbage collection to limit the program's heap usage. Using this mechanism we can get performance either close to or better than the best possible with a virtual-machine controlled heap size, independent of the garbage collector, the physical memory size, and the compiler optimization. In the extreme case, the adaptive heap sizing leads to a factor of 7.8 overall speedup over the best possible single heap size, when both are using the same garbage collector.

		pJBB 192M opt	pJBB 128M opt	pJBB 128M base	ipsixql 128M opt	ipsixql 128M base	compress	jess	db	mtrt	AVG
GenCopy	best	-30%	26%	-0%	-30%	-10%	-16%	-39%	9%	13%	-9%
	2nd	-27%	34%	2%	-15%	-9%	-15%	-30%	20%	80%	4%
	3rd	-24%	37%	3%	-11%	-8%	11%	-7%	32%	85%	13%
	default	-28%	274%	20%	129%	17%	26%	9%	13%	118%	64%
CopyMS	best	-8%	14%	-14%	-45%	-14%	-3%	2%	3%	-28%	-10%
	2nd	-7%	22%	-12%	-39%	-14%	-1%	6%	10%	159%	14%
	3rd	-6%	45%	-2%	-37%	-14%	3%	168%	12%	571%	82%
	default	-2%	648%	100%	-37%	-0%	28%	2925%	40%	1016%	524%
Mark-Sweep	best	-6%	780%	-7%	-23%	-14%	4%	262%	52%	-35%	113%
	2nd	-6%	791%	4%	-9%	-12%	13%	725%	53%	369%	214%
	3rd	-6%	835%	28%	-4%	-10%	18%	748%	54%	381%	227%
	default	-5%	1603%	436%	1368%	-2%	40%	861%	92%	584%	553%

Table 3. The speedup of the adaptive scheme over the performances of the best fixed, the second best fixed, the third best fixed and the default VM-controlled heap sizes for the three garbage collectors. A negative number means a slowdown.

7. Related Work

Several recent studies examined methods by which a program (and not the JVM) controls when the heap is collected, and what part of the heap to collect. Buytaert et al. use offline profiling to determine the amount of reachable data as the program runs, and then generate a listing of program points to indicate when collecting the heap will be most profitable. At runtime, they then can then collect the heap when the ratio of reachable to unreachable data is most favorable [5]. Similar work by Ding et al. used a Lisp interpreter to show that limiting collections to occur only at phase boundaries reduced GC overhead and improved data locality [9]. We expand upon these past studies by including information from the application, VM, and operating system to guide our collection decisions and select collection points that both minimize the amount of reachable data *and* maximize the use of available memory.

Soman et al. used a modified JVM to allow a program to select which garbage collector to use at the program load time based on profiling and user annotation [20]. The control mechanism was applied before the start of the program, and the heap size was fixed rather than adaptive. Other studies have proposed methods by which the JVM adapts the heap size to improve performance. Several of these approaches, like ours, were focused on reducing paging costs. Alonso and Appel presented a collector which reduced the heap size when advised that memory pressure was increasing [1]. Yang et al. modified the operating system to use an approximate reuse distance histogram to estimate the current available memory size. They then developed collector models that enabled the JVM to select heaps size that fully utilize physical memory [22]. Hertz et al. developed a paging-aware garbage collector and a modified virtual memory manager that cooperated to greatly reduce the paging costs of large heaps [11]. While these past schemes require modifications to the virtual machine and, for all but one, the operating system, our approach runs only at the application-level and does not need these intrusive modifications.

Our technique depends on recurring phase behavior in a program. Many phase detection techniques have been proposed. These algorithms exploit aspects of the program structure such as loops and procedures [4, 10, 13, 16], regions [12], and call sites [15] to determine phase boundaries. Most techniques use fixed thresholds to select coarse-grained, recurring phases. Georges et al. selected Java methods whose behavior variation was relatively small [10]. Unfortunately the phase behavior of the utility programs on which we focus is dependent upon the program inputs, and may not be captured by static analysis. We rely upon the active profiling techniques we developed in earlier work that use regular inputs to induce behavior repeatable enough for analysis and phase marking [18]. This analysis can be performed on low-level code including program binaries.

While heap management adds several new wrinkles, there is a long history of work on creating virtual memory managers that adapt to program behavior to reduce paging. Smaragdakis et al. developed early eviction LRU (EELRU), which made use of recency information to improve eviction decisions [19]. Last reuse distance, another recency metric, was used by Jiang and Zhang to prevent interactive applications from thrashing. Chen et al. [6] also used last reuse distance to reduce the total amount of paging [14]. Zhou et al. tracked reuse distance histograms for each fixed time interval to improve the throughput and response time of multiple processes through selective memory allocation [23]. All of these techniques try finding the best subset of the working set to keep in physical memory, but are of limited benefit when the working set fits entirely in available memory. Heap management, on the other hand, can improve performance when given additional physical memory by increasing the heap size, thus reducing the frequency of garbage collection. Our program-level adaptive memory management system tries balancing these costs by choosing heap sizes that minimize the costs of smaller heap sizes (more frequent garbage collections) and of larger heap sizes (increased paging activity).

Acknowledgments

We are grateful to IBM Research for making the Jikes RVM system available under open source terms. The MMTk memory management toolkit was particularly helpful.

References

[1] R. Alonso and A. W. Appel. Advisor for flexible working sets. In *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 153–162, Boulder, CO, May 1990.

[2] E. Andreasson, F. Hoffmann, and O. Lindholm. To collect or not to collect? Machine learning for memory management. In *JVM '02: Proceedings of the Java Virtual Machine Research and Technology Symposium*, August 2002.

[3] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *Thirtieth*, New Orleans, LA, Jan. 2003.

[4] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Pro-*

ceedings of the 33rd International Symposium on Microarchitecture, Monterey, California, December 2000.

[5] D. Buytaert, K. Venstermans, L. Eeckhout, and K. D. Bosschere. Garbage collection hints. In *Proceedings of HIPEAC'05. Lecture Notes in Computer Science Volume 3793, Springer-Verlag*, November 2005.

[6] F. Chen, S. Jiang, and X. Zhang. CLOCK-Pro: an effective improvement of the CLOCK replacement. In *Proceedings of USENIX Annual Technical Conference*, 2005.

[7] S. P. E. Corporation. Specjbb2000. `http://www.spec.org/jbb2000/docs/userguide.html`.

[8] S. P. E. Corporation. Specjvm98 documentation, Mar. 1999.

[9] C. Ding, C. Zhang, X. Shen, and M. Ogihara. Gated memory control for memory monitoring, leak detection and garbage collection. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Memory System Performance*, Chicago, IL, June 2005.

[10] A. Georges, D. Buytaert, L. Eeckhout, and K. D. Bosschere. Method-level phase behavior in Java workloads. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, October 2004.

[11] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference On Programming Language Design and Implementation*, pages 143–153, New York, NY, USA, 2005. ACM Press.

[12] C.-H. Hsu and U. Kremer. The design, implementation and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[13] M. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. In *Proceedings of the International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[14] S. Jiang and X. Zhang. TPF: a dynamic system thrashing protection facility. *Software Practice and Experience*, 32(3), 2002.

[15] J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. Technical Report CS2004-0804, UCSD, November 2004. *conference version to appear in CGO'06*.

[16] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Proceedings of the International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[17] J. E. B. Moss, K. S. McKinley, S. M. Blackburn, E. D. Berger, A. Daiwan, A. Hosking, D. Stefanovic, and C. Weems. The dacapo project. Technical report, 2004.

[18] X. Shen, C. Ding, S. Dwarkadas, and M. L. Scott. Characterizing phases in service-oriented applications. Technical Report TR 848, Department of Computer Science, University of Rochester, November 2004.

[19] Y. Smaragdakis, S. Kaplan, and P. Wilson. The EELRU adaptive replacement algorithm. *Perform. Eval.*, 53(2):93–123, 2003.

[20] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, New York, NY, USA, 2004. ACM Press.

[21] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[22] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic heap sizing: taking real memory into account. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 61–72, New York, NY, USA, 2004. ACM Press.

[23] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, USA, October 2004.