Waste Not, Want Not: Adaptive Garbage Collection in a Shared Environment

Chengliang Zhang[†], Kirk Kelsey[†], Xipeng Shen[‡]
Chen Ding[†], Matthew Hertz[⋆], and Mitsu Ogihara[†]

[†]Department of Computer Science
University of Rochester
{zhangchl,kelsey,cding,ogihara}@cs.rochester.edu
[‡]Department of Computer Science
The College of William and Mary
xshen@cs.wm.edu
[⋆]Department of Computer Science
Canisius College
hertzm@canisius.edu

University of Rochester Computer Science Deptment Technical Report TR-2006-908

Abstract

Limiting the amount of memory available to a program can hamstring its performance, however in a garbage collected environment allowing too large of a heap size can also be detrimental. Because garbage collection will occasionally access the entire heap, having a significant amount of virtual memory becomes expensive. Determining the appropriate size for a program's heap is not only important, but difficult in light of various virtual machines, operating systems, and levels of multi-programming with which the program may be run.

We present a model for program memory usage with which we can show how effective multi-programming is likely to be. In addition, we present an automated system for adding control at the program level that allows runtime adaptation of a program's heap size. The process is fully automatic and requires no extra coding on the part of programmers. We discuss two adaptive schemes: the first acts independently, and while performing competitively, the system behaves politely in a multi-programmed environment. The second scheme explicitly cooperates when multiple instances are running. Both schemes are evaluated in terms of their response time, throughput, and fairness.

1 Introduction

Software developers are taking advantage of garbage collection for the many engineering benefits it provides using either garbage-collected languages such as Lisp, ML, and Java or conservative garbage collectors such as the Boehm-Demers-Weiser collector [6]. While a conventional program uses exactly as much memory as it needs, the memory use of a garbage collected program can expand beyond its need. This difference raises the possibility of *resource-based garbage collection*, which is to adapt the frequency of GC in response to the changing amount of available memory.

Memory usage depends first on the program demand. The heap must be large enough to accommodate all reachable data. When the heap size is large enough, setting a larger heap size in the virtual machine will lead to fewer calls to the collector, which means lower collection time, so long as the heap size does not exceed the available memory and cause paging. The common scheme for Java programs is to use a fixed range by which the heap size can exceed the program's need. As an example Jikes RVM uses a range of 50MB to 100MB by default. This is often a mismatch to the available resource for a number of reasons. First, the actual memory usage depends on the garbage collector implementation (e.g. twice the heap size for a copying collector), not counting the memory needed by the virtual machine and the operating system. Second, knowing the exact amount of available memory is difficult because it requires ascertaining the active memory usage of other programs. Much memory may be occupied but not used (for example the file cache) and can be made available. Last but not least, the heap size, which is set before the execution, cannot respond to change of conditions in the middle of an execution.

In this paper we address the problem of resource-based garbage collection in a shared environment, which is increasingly common on today's multi-processor, multi-core, and multi-threaded machines. Garbage-collected programs, however, lack a firm basis for high degrees of multi-programming if two problems remain unsolved. The amount of available memory may change significantly and dynamically because of memory sharing, making any setup vulnerable to extreme and adverse conditions if it does not adapt to the dynamic resource. Also, the moment a program starts to adapt, it embarks on a collision course with other like minded programs, no matter how much memory is available.

We first revise the classical performance model, which is demand-based, to accommodate resource-based memory management. We introduce a new quantitative notion called *time-memory curve* and use it to predict the performance of memory sharing in this new context. An example of a resource-based garbage collection system is *program-level adaptive memory management (PAMM)*, which uses run-time monitoring and heap-size control for resource adaptation [24]. We develop a scheme similar to PAMM for memory sharing. While the previous scheme used semi-manual phase analysis, we make it fully automatic so it can be applied to general programs. The previous scheme is designed for a static environment and by nature is *selfish*. It does not consider other contenders for the monopoly on memory. We present *cooperative PAMM*, which dynamically divides the available memory among the resource contenders through a shared white-board data structure to maximize system throughput.

We evaluate the new performance model and resource-based adaptive GC using three large Java programs, the Jikes RVM, and dual-processor PCs. For the initial comparison we use two and four identical processes and compare the finishing time between the default GC and the resource-based GC and between selfish PAMM and cooperative PAMM. We compare the two schemes based on their throughput, response time and fairness. We also measure the accuracy of the time-memory curve in predicting the performance of memory sharing.

The adaptive GC techniques are implemented at the program level by inserting a run-time monitor and controller, without users' effort. They monitor the program demand and the memory performance (in particular the memory paging) to adapt the memory demand with the available resource. They do not require changes to the virtual machine or the operating system, so they can be automatically deployed on existing systems.

2 Memory performance model

Intuitively more memory leads to better performance. The goal of a model is to predict what amount of memory can improve performance for which programs and by how much, in other words, what is the memory locality of a system.

The classical model of memory performance as pioneered in work done by Mattson et al., Denning, and Smith [10, 15, 19] assumes that a program's memory demand (including the heap size) does not change with the environment. Here we revisit the classical model of memory performance, explain the difference between demand-based and resource-based memory allocation in formal terms, and show how a new model (the time-memory curve) predicts the performance of garbage-collected systems.

2.1 The classical model

Let t^{total}, $t^{computing}$, and t^{paging} be the total running time of a program P, its computing time, and its paging time respectively. t^{total} is at least the sum of the latter two, and the equality holds in the ideal case, i.e., when the resident memory size stays constant and the costs of scheduling, context switches, and cache sharing are negligible comparing to the cost of computation and paging.

Model 1 (Classical model) *The following inequality holds in the classical model.*
$t^{total} \geq t^{computing} + t^{paging}$*, and*
$t^{paging} = f(\Phi(M))$

where the function $\Phi(M)$ gives the number of page faults for a given amount of physical memory M, and the function f gives the time needed for the page faults. The first term, the pure program cost t^{cpu}, is not affected by the memory size.

For decades, the histogram of stack distances has been the standard model for Φ since Mattson et al.'s seminal work in 1970 [15]. The model can be viewed as a *fault-memory curve*, which plots the number of faults with respect to the memory size. This curve has long been used in virtual memory and cache performance studies, as pioneered by Denning, who showed the knee of memory locality [10], and by Smith, who measured the effect of set-associative memory [19].

The classical model is not precise for interpreted programs for three reasons. First, M is not easy to measure because a program shares memory with the virtual machine. Second, the number of page faults does not translate directly to time. The cost of loading a page differs from page to page due to the disk layouts, the effect of prefetching and disk caching. As a result, f may not be monotonic, that is, the fewer the page faults, the lower the total paging time. The last and perhaps most significant problem is that a GC-based program can change its run-time memory demand by adjusting its heap size. It cannot be modeled by the stack algorithms of Mattson et al. if the memory allocation depends on the memory size (see pp. 93 of [15]).

2.2 The time-memory curve

To model GC-based programs, we redefine the computing and paging time with a new parameter for the size of the heap, H as below.

Extension 1 (Heap performance model)

$$t^{computing} = t^{mutator} + t^{gc}(H)$$

$$t^{paging} = f(\Phi^{gc}(H, M))$$

where $t^{mutator}$ is the mutator time, $t^{gc}(H)$ is the GC time with heap size H, H, and $\Phi^{gc}(H, M)$ is the number of page faults for heap size H and memory size M.

The classical model did not consider the heap size because most traditional programs either avoid dynamic memory allocation (for example in Fortran 77 programs) or use manual memory allocation. In either case the heap size depends completely on the program, so it can be considered implicitly as part of the program locality. The data layout is also a factor, but it is present in all cases so we do not consider it explicitly.

For programs that use garbage collection, the heap usage depends on memory allocation by the virtual machine. In current systems, a default heap size, $H_{default}$ is used, and a garbage collector is invoked when the heap is full. Part of the default scheme is a provision for increasing the heap limit if it is too small to hold the reachable data needed by the program. We call this scheme *Demand-based GC.*

Extension 2 (Demand-based GC)

$$H^{demand} = gc(P, H_{default})$$

The heap size is initialized by a parameter $H_{default}$ and may increase if the size is too small for the reachable data used by the program P.

In demand-based GC, the heap size does not adapt to the amount of available memory M, at least not during execution.

The PAMM scheme described in Section 3 monitors the heap usage and grows the heap limit until it uses all available memory. As a result, the heap size depends on the size of the available memory M rather than a fixed starting point like $H_{default}$. We call schemes like PAMM *resource-based GC*. Although the name emphasizes the resource aspect, resource-based GC actually considers both the demand and the resource as shown in the formula below.

Extension 3 (Resource-based GC)

$$H^{resurce} = gc(P, M)$$

The heap size is automatically determined by the program demand P and the available memory M.

If we fix the program P and the garbage collector gc (removing them as variable parameters), the memory time of resource-based schemes like PAMM is a function of only the memory size M. We represent it as $t^{resource}(M)$. In demand-based schemes, H depends on an extra variable $H_{default}$, so the total time is a function of the default heap size and the memory size, which we represent as $t^{demand}(M, H_{default})$. The overall performance of the two schemes are as follows.

Model 2 (Time-memory model)

$$\begin{aligned}
t^{resource}(M) &= t^{mutator} + t^{gc}(H^{resource}(M)) + \\
&\quad f(\Phi^{gc}(H^{resource}(M), M))) \\
&= t^{mutator} + t^{gc}(M) + f(\Phi^{gc}(M) \\
t^{demand}(M, H_{default}) &= t^{mutator} + t^{gc}(H(H_{default})) + \\
&\quad f(\Phi^{gc}(H^{demand}(M, H_{default}))) \\
&= t^{mutator} + t^{gc}(H_{default}) + f(\Phi^{gc}(M, H_{default}))
\end{aligned}$$

In the abstract form, the performance of demand-based GC is a function of M and $H_{default}$, while the performance of resource-based GC is a function of only M, because the heap size is adjusted based on M. Implied by this difference is that the execution time for the adaptive scheme is unique for each memory size, but it differs in the fixed scheme for different $H_{default}$ even for the same memory size.

We note that the resource-based GC becomes demand based in the extreme case when the available memory is not large enough to hold the reachable data of a program.

While it is possible for a user to minimize t^{demand} by finding the best $H_{default}$: the manual control is difficult for three reasons. First, the actual memory demand depends on the garbage collector and may be much more than $H_{default}$. Second, finding the exact amount of available memory requires ascertaining the active memory usage of the operating system, the virtual machine, and other concurrent programs. Finally, $H_{default}$ is set at the beginning and cannot respond to a change of conditions in the middle of an execution. While in theory a user can choose a good $H_{default}$ knowing the physical memory size on a machine, in reality the task is beyond the grasp of all but the most proficient and ardent users. Sometimes it happens that the best fixed heap size does not perform as effectively as the adaptive scheme because the latter may use different heap sizes in the same run [24]. To find the minimal $t^{default}$, in the worst case, a manual or automatic search needs to check every $h_{default}$, which takes $O(M)$ number of runs for each memory size M.

In Figure 1 we can see the danger of selecting the wrong heap size. The valley in the graph around a heap size of 100M represents a good choise for those physical memory sizes. If the heap size deviates from this value, the results degrade rapidly. For larger physical memory sizes, the valley plains out and the choice of heap size becomes somewhat less critical. This has two important implications for the shared environment. First, as the available memory changes dynamically in a shared environment, demand-based memory management may incur an extremely high penalty. Second, even for an adaptive scheme that can find the best heap size, a parallel execution may not improve the throughput when there is not enough memory to share. For example for the memory size 112MB, the program with the best fixed heap size takes almost three times as long as the program would with twice the memory. Therefore, running two copies sequentially using one processor is 40% faster than running the same two copies in parallel using two processors.

2.3 To share or not to share

In a shared environment, the throughput, response time, and fairness depend on the memory allocation among concurrent applications. One may run multiple applications sequentially so each run has the exclusive use of the entire memory, or one may let some of them run in parallel and share the available memory. The best throughput is easiest to define—the execution that finishes all applications in the least amount of time. The time-memory curve can make quantitative predictions. Consider the basic case below.

Model 3 (Concurrent execution) *A concurrent execution for N programs running on N identical processors with a total amount of memory M finishes no slower than running those N programs sequentially if $\exists m_1, m_2, ..., m_n$ such that*

$$\max_{i=1...N} t_i^{resource}(m_i) \leq \sum_{i=1}^{N} t_i^{resource}(M)$$

where $\sum_{i=1}^{N} m_i \leq M$.

A corollary to our model is that when all programs are identical, the concurrent execution finishes no slower the sequential run if

$$t^{resource}(M/N) \leq N t^{resource}(M)$$

This model does not consider the sharing of a single processor because the overlap of computation and paging depends on the program and cannot lead to a better performance than running each program on a processor. We also

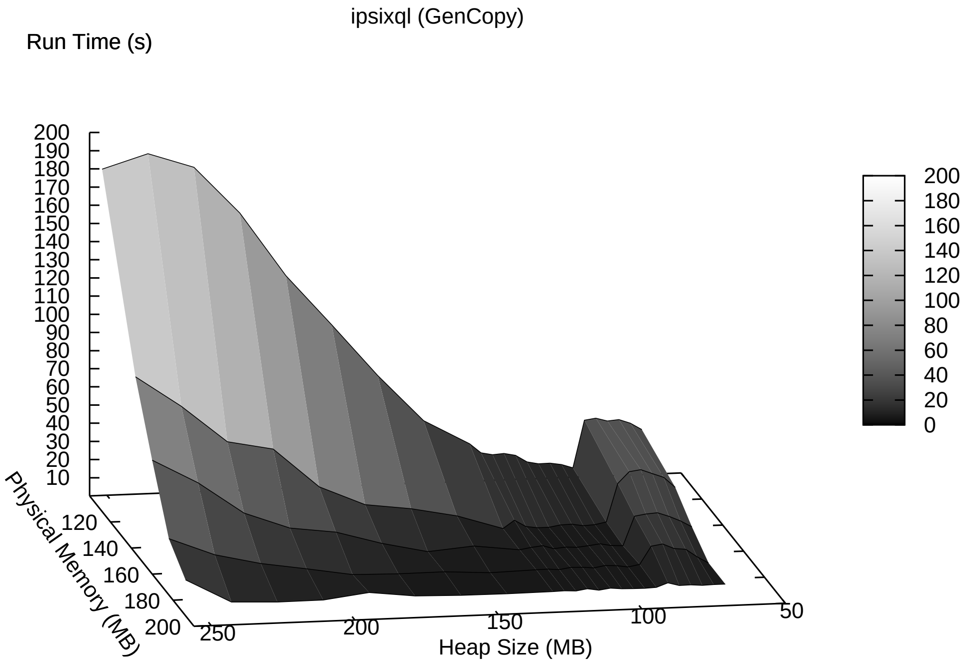


Figure 1: Running time as a result of physical memory and heap sizes

assume no I/O contention from paging of multiple processes. With these assumptions, the time-memory curve can be used to predict the effect of memory sharing. As a result, it can be used to find the multiprogramming scheme with the best throughput.

The time-memory curve solves (to the extent stated just now) two new problems of memory management caused by GC-based languages such as Lisp and Java. First, a garbage-collected program can always use all available memory no matter how large it is (no larger than the address space of course), while the memory usage of conventional programs is independent of the available memory. Second, resource-based GC cannot be modeled by stack algorithms because its memory allocation depends on the memory size (see pp. 93 of [15]), so the standard performance model is no longer valid.

3 Resource-based Garbage Collection

In this section we present our approach toward resource-based garbage collection. We first provide a review of the program-level adaptive memory management approach (*PAMM*) that we developed in earlier work [24]. Then, we introduce our approach which automatically converts existing programs to run in our resource-based garbage collection scheme.

Our previous definition of PAMM is selfish with processes using only their observations of their own state to adjust the heap so that it uses the available resources. In comparison with this selfish behavior, *cooperative PAMM* allows processes to share their observations and make global descisions with a goal of better utilizing resources.

3.1 Selfish PAMM

PAMM is a program-level adaptive memory management control scheme that makes existing demand-driven garbage collectors behave in a resource-based manner. Runs using PAMM begin by requesting that the virtual machine provide the application with an extremely large heap. While the program runs, it periodically polls the PAMM memory controller. The PAMM controller uses both the current heap size and a count of the application's page faults since its last collection to determine if the heap should be collected. When the controller determines that growing the heap will reduce program throughput (either because it sees signs of increasing memory pressure or past data shows the current heap size to be optimal), it prompts the application to explicitly collect the heap.

The information required by the PAMM controller can be easily and efficiently obtained. The Linux operating system already tracks page fault counts on a per-process bases and makes this information available via the `/proc/self/stat` pseudo-file. As part of the `/proc` system, the `stat` file does not really exist, but serves as an interface to the data structure the kernel maintains in memory. The current heap size is normally tracked by the garbage collector and many languages, e.g. Java, include requests for the current heap size within their API.

A key feature of the PAMM controller is its use of a *soft* bound on the heap size. Rather than only using the virtual machine's heap size limits (the *hard* bound), our program-level system uses and adjusts this soft bound to allow the heap to fully utilize available memory, but not to increase to a size that would trigger more paging.

The PAMM controller adjusts the soft bound using two new variables: left mark and right mark. The left mark serves as a lower limit on the soft bound and the right mark is an upper soft bound limit. As the controller adapts its heap size, it makes changes at a predefined granularity. We made changes at a 10MB step size in these experiments.

At the start of the program, we set the left mark to our initial heap size and the right mark as the initial heap size plus two step sizes (e.g., 20MB for our experiments). Following each garbage collection, the PAMM controller examines the current heap size and the number of page faults that occured during that collection.

When there are few page faults, PAMM knows that it can allow the heap to expand safely. In this case, the controller uses the current soft bound as the new left mark and selects a new soft bound equal to the mean of the old left mark and the right mark. It is possible, however, that the right mark could be smaller than the current optimal heap size. If the soft bound grows by under 1MB, then PAMM artificially inflates both its new soft bound and right mark by 1MB. This ensures that it will continue trying larger heap sizes, but does so conservatively.

Collections that trigger a large number of page faults inform the controller that the heap is too large for available memory; it must decrease its soft bound. Shrinking the heap is done using a similar method as growing the heap. In this case, the controller updates the right mark to the current soft bound. The soft bound is then set as the average of the old right mark and left mark. If the change to the soft bound is less than 1MB, PAMM forces the soft bound and left mark to decrease by at least 1MB.

This process leaves open the question of when the controller should grow or shrink the heap. PAMM defines a page fault threshold (we use a threshold of 10). If, upon reaching the soft bound, the system has had fewer page faults than this threshold, then the heap may not be using all of the available memory and PAMM will try increasing the soft bound. If, even before the heap grows to the soft bound, PAMM detects the application has triggered more page faults than the threshold, then the heap is too big and we must immediately collect the heap and decrease our soft bound. In this manner our binary search exploration of heap sizes adapts to find optimal heap sizes.

Automating PAMM

In previous work, introducing adaptive memory management at the program level requires the programmer to first identify phases semi-automatically. Monitoring and adjusting the heap size can then be instrumented automatically. Later experiments show that phase analysis is not the only choice to monitor the memory and page faults. In this work, we provide a fully automatic mechanism for PAMM.

The first step of automatic PAMM is to instrument the Java program. The programmer needs to specify a list of class files including the class that contains the main method used to invoke the program. The instrumentation identifies each allocation point and wraps the allocation with SmartGC functionality. The SmartGC system maintains a counter, which is incremented at each allocation. The system checks the current heap size every one thousand allocations and the page faults every 256K memory allocated. A garbage collection is triggered if necessary.

Using this instrumentation approach allows the PAMM system to control the heap size without detailed information about the program design. By relying solely on the allocation behavior of the program at run-time, the programmer does not need to specify how they expect the program to behave.

The controller could be implemented in a virtual machine. Taking the Jikes virtual machine as an example, we could monitor the memory usage and page faults at the poll function of each garbage collector. In this paper, we show an implementation at the program level.

3.2 Cooperative PAMM

In comparison with PAMM's selfish approach of having processes adapt to the available resources, we developed a new version of our program-level adaptive memory manager that enables processes to cooperatively manage resources and ensure the equitable sharing of memory. As in the selfish implementation of PAMM, the cooperative PAMM uses page fault counts to adjust the heap size to utilize available memory fully. The cooperative PAMM, however, evaluates the aggregate number of page faults for all of its processes and, when needed, adjusts their heap sizes equally. This cooperative approach should allow processes to grow or shrink their heap at the earliest possible time, while also ensuring that one of the applications will not grow its heap at the expense of the others.

As with selfish PAMM, applications periodically report both their heap size and page fault counts to the cooperative PAMM controller. Applications begin this process by reporting their current heap size to the controller, by writing this value to the *whiteboard* the cooperative PAMM controller maintains. As with selfish PAMM, the controller first checks if the heap size exceeds the soft bound and must therefore collect the heap. If the heap is still within the limits of the controller's soft bound, but has grown beyond a minimum threshold (for these experiments we used a threshold of 256KB), the controller then checks the aggregate number of page faults since the application last collected the heap. When this aggregate count shows that the system is paging, e.g., when the count is above a low minimum threshold, the controller directs the application to collect the heap. Because the controller is checking the number of page faults since the current process last performed garbage collection, different applications may have different page fault counts and have garbage collection triggered at different times. If the controller determines that a collection is not needed, the application continues to execute the program as before.

The cooperative PAMM controller adjusts its soft bound in a manner similar to the selfish PAMM controller. When memory is plentiful, garbage collections are triggered by the heap growing to the controller's soft bound. In these situations, the cooperative controller increases the soft bound to utilize more of the available memory. When collecting the heap causes more page faults than the page fault threshold, the controller shrinks its soft bound to avoid futher paging. Unlike the selfish PAMM controller, however, cooperative PAMM increases or decreases the amount of memory it is using by a consistent step size of 10MB and the soft bound by this step size divided by the current number of processes it is controlling.

4 Fairness and response time

While optimizing throughput in a multi-programmed environment is difficult enough, there are other dimensions that a user may wish to evaluate a system. While not traditionally considered in this research, users may also wish to consider issues of fairness (how evenly applications progress) and responsiveness (how quickly an application react to user input). This multi-dimensional evaluation is all the harder because improvements on one dimension often means degrading performance along other dimensions. We now discuss how we gather data to evaluate these additional dimensions and then present several examples documenting why it is important to consider these details.

Model 4 (Fairness) *Fairness shows how evenly applications progress. We define fairness as:*

$$Fairness = \min_{i=1\ldots N} \frac{Share^{i}}{Share^{ideal}}. \tag{1}$$

$Share^{ideal}$ *is the ideal time spent on each application in a system with* N *applications running on* m *different processors.* $Share^{i}$ *is the actual fraction of total time that an application* i *spends on making progress. Formally, they can expressed as:*

$$Share^{ideal} = \frac{m}{N}, \tag{2}$$

$$Share^{i} = \frac{t_i^{computing} - t_i^{gc}}{TotalTimeElapsed} \tag{3}$$

Model 5 (Responsiveness) *Responsiveness shows how quickly an application reacts to user input. We could define a systems responsiveness as the longest time for any application between moments where the application is able to make progress.*

To generate these metrics, we only need access to the *Total Time Elapsed* and t_i^{cpu}, which are available via `/proc/self/stat` pseudo-files, and t_i^{gc} which can be gathered from the Java virtual machine. In our experiments, runs of our system computing and recording these data require the same time as identical experiments that do not perform this computation (e.g., the difference in execution times is less than our measurement error).

To better understand the relationship between these dimensions of measurement we present several idealized fairness curves. Figure 2 presents idealized data as a means of exploring these trade-offs. We present these fairness curves in a manner similar to bounded memory utilization, or BMU, curves. BMU curves, first proposed by Sachinran, et

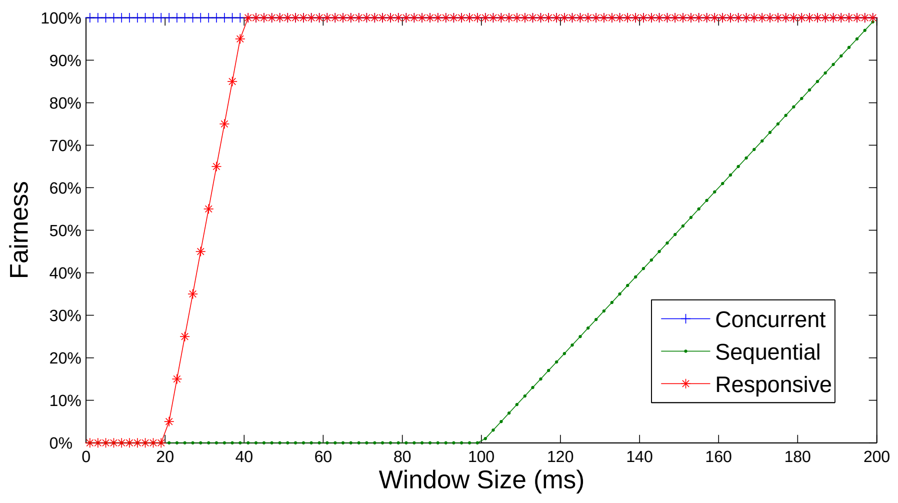


Figure 2: Idealized fairness curves highlighting different extreme behaviors. The curve labeled "Concurrent" shows a run with sufficient resources execute all applications concurrently. The curve labeled "Sequential" shows a run which executes the applications sequentially. The curve labeled "responsive" shows a curve which cycles through the applications executing to minimize response times. Curves to the left and higher are better.

al. [17], are often used to help evaluate garbage collection pause times. Constructing the fairness graphs similarly makes understanding them easier. Each point in the fairness graph shows the minimum fairness for any window of the given size or larger. The rightmost point plotted on the graph represents the run's total throughput – the total elapsed time needed to execute the applications. A curve's x-intercept represents the runs responsiveness – the longest delay any application spent making progress. These graphs therefore illustrate all three dimensions we wish to examine.

The three curves in Figure 2 present extreme cases optimizing throughput, fairness, and responsiveness. The curve labeled "concurrent" illustrates a run without any resource bound. Assuming all programs could execute concurrently, the run would record a perfect fairness value of 1 at all window sizes. The curve labeled "sequential", however, shows a run of 2 programs executed sequentially for maximum throughput. This curve shows the run is barely responsive (the second application waits until after the first completes). While the fairness also suffers, if the run were limited by a single CPU, it would ultimately achieve perfect fairness (at the end of the run each application receives one-half of the time to make progress). The final curve, labeled "responsive", illustrates a run maximizing reponsiveness by cycling through the applications and giving each a single quanta of progress. This curve is therefore similar to "concurrent", but the fairness is limited to however many applications can be executed before resources are fully consumed. In addition, the "responsive" curve's responsiveness equals the time needed to cycle all of the applications into memory.

5 Results

To evaluate cooperative PAMM, we compare its performance with the default heap sizing approach within the Jikes RVM. We also compare an implementation of selfish PAMM using 3 of the garbage collection algorithms (GenMS, GenCopy, and CopyMS) included within MMTk. We describe the collectors we use and the benchmarks on which we run our collectors in more detail below.

5.1 Methodology

We performed our experiments on two types of Linux machines. One type has two 1.26GHz Pentium III processors with 1G swap space. Another type has two hyperthreaded 2GHz Intel Xeon processors with 2G swap space. All of the machines have 2G memory and 512K cache for each processor. To create the memory pressure on garbage collection, we use a small program to explicitly lock a part of the memory. We repeat each experiment three times and use the one with shortest running time.

In our experiments, we choose three programs with significant memory footprints as benchmarks: pseudoJBB [9],

Benchmark	#bytes	#allocations	param. (Fir.)	param. (Sec.)	mini. heap
pseudoJBB	0.92G	28.69M	10000	350,000	42M
ipsixql	2.64G	90.64M	5-7-1	1-7-1	19M
lucene	1.19G	42.47M	200	2000	17M

Table 1: Benchmarks and their parameters

ipsixql and lucene [2]. The *pseudoJBB* benchmark simulates a warehouse system which repeatedly processes 6 different types of transactions. It is modified from SPECjbb benchmark to perform only a fixed number of random loads with a single thread. It is widely considered to be representative of a server load. In our experiment, we tested 350,000 transactions. *Ipsixql* is an XML database program from the DeCapo benchmark suite. We tested all of the 7 given queries 5 times each. Benchmark *lucene* is a high-performance and full featured text search engine library. We first index over the 20 newsgroups [1], which is a collection of approximately 20,000 newsgroup documents, partitioned (nearly) evenly across 20 different newsgroups. The index file takes 16.9M. We created a set of single word queries, 90% of which are words drawn randomly from the original dataset and the remainder are words known not to be in the index. For each application, we instrument all of memory allocation instructions using Soot [21]. Table 1 shows the information of each benchmark. In the table, the number of bytes means the total amount of memory that is allocated by the application except the virtual machine.

All of the benchmarks are compiled using the Jikes compiler (version 1.22) and executed in the Jikes RVM (version 2.4.0). We follow the second run approach which was first proposed by Bacon et al. [5] to exclude the time spent on optimization. For the first run, we use the system's default setting. We set the heap size to be fixed (1G) for the second run, whose result is used for the comparison of different schemes. To duplicate the same environment for all of the experiments, we adopt a pseudo-adaptive method [13, 16] to create the same optimized code. It is to first generate an advice file and then use the advised optimizations in the second run. Since the first run is only to optimize the code, we use a much smaller input as shown in table 1.

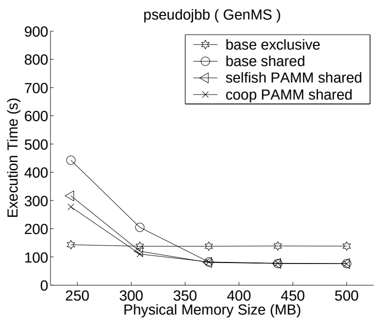
We compare the different schemes of heap sizing over the three representative garbage collectors: *GenMS*, *GenCopy* and *CopyMS*. All of them are "stop the world" approaches. *GenMS* is a generational scheme in which a mature space is managed using Mark-Sweep. In the Jikes implementation, the nursery 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 if the nursery size falls below a static threshold. GenCopy is nearly identical to GenMS, except that the mature space is managed using a copying collector. 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 marked and later copied into the second space. No write barrier is present, so every collection is performed over the whole heap.

5.2 Cooperative PAMM vs. the Base System

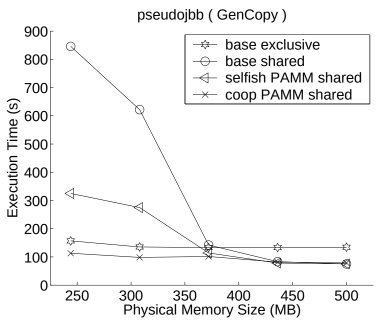
One key factor in analyzing cooperative PAMM's performance is to consider its throughput relative to the default behavior of the Jikes RVM system. For the results labeled "Base Shared", two instances of the base system execute the benchmark simultaneously and therefore must share resources. As another point of comparison, we also evaluate the results when the two runs are executed sequentially so that a process has exclusive access to the resources. We label these latter experiements as "Base Exclusive." Figures 3(b) through 3(f) show all the execution time and Figures 4(b) through 4(f) show the page faults.

Comparing Base Shared and Base Exclusive runs shows the limitations of demand-based fixed garbage collection. When physical memory is plentiful (the right-hand side of these Figures), there is little contention even while sharing resources. With both executions fitting entirely in physical memory, the Base Shared runs complete well before the Base Exclusive runs. When physical memory is limited, however, the two concurrent runs fight over the available memory and end up triggering a large number of page faults. As a result of all these page faults, the Base Exclusive runs can require as little as one-fourth of the time needed by Base Shared. In fact, Figure 3(i) is the only experiment where the lines for Base Shared and Base Exclusive do not cross. These runs, however, needed very little memory and triggered very little paging. When paging may arise, however, there is not a static solution for demand-based garbage collection.

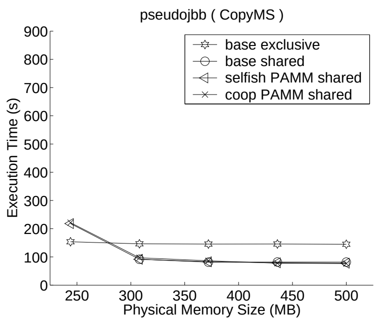
Coperative PAMM performance compares quite favorably with Base Shared. One example of this can be seen



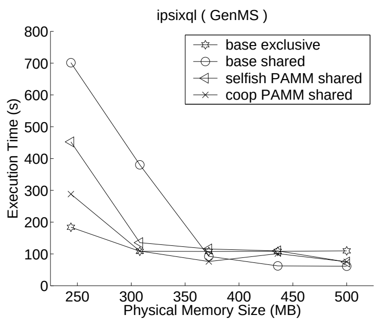
(a) GenMS running pseudoJBB



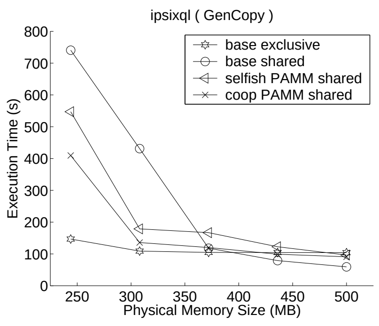
(b) GenCopy running pseudoJBB



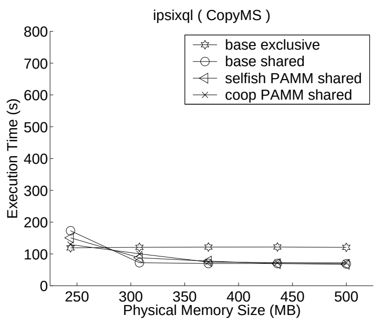
(c) CopyMS running pseudoJBB



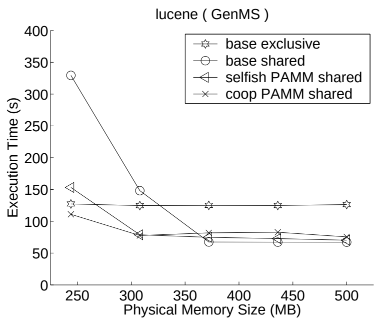
(d) GenMS running ipsixql



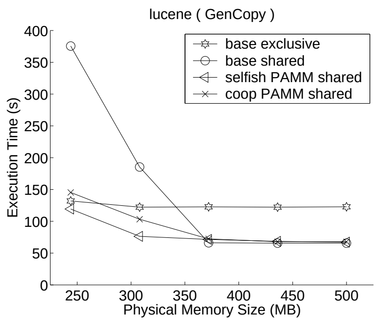
(e) GenCopy running ipsixql



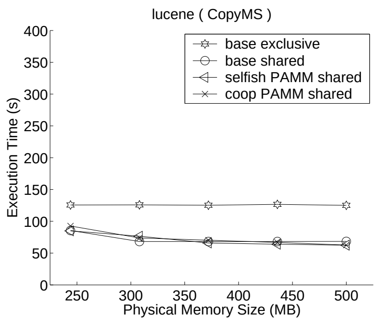
(f) CopyMS running ipsixql



(g) GenMS running lucene

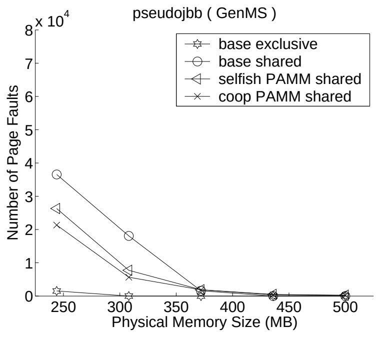


(h) GenCopy running lucene

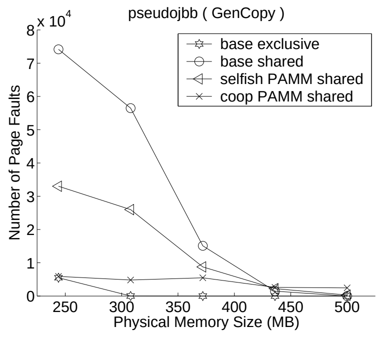


(i) CopyMS running lucene

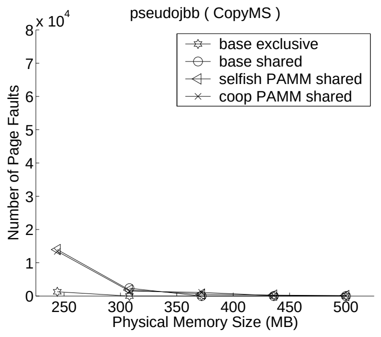
Figure 3: Graphs showing the total execution time when varying the amount of physical memory on the machine.



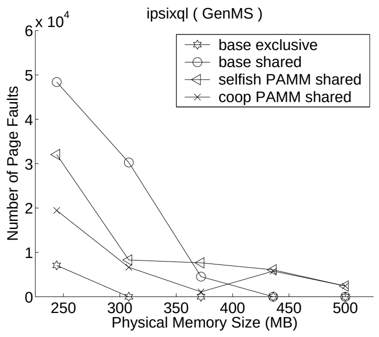
(a) GenMS running pseudoJBB



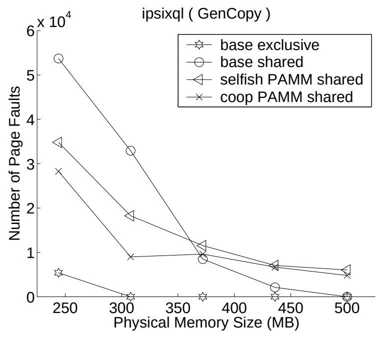
(b) GenCopy running pseudoJBB



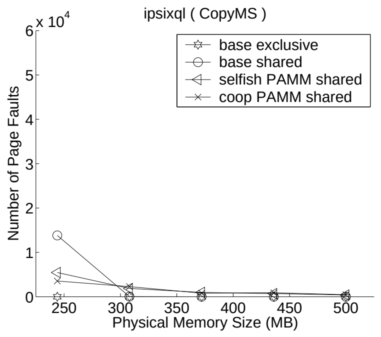
(c) CopyMS running pseudoJBB



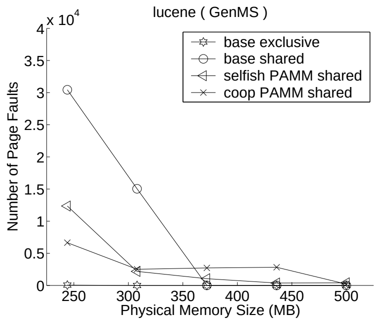
(d) GenMS running ipsixql



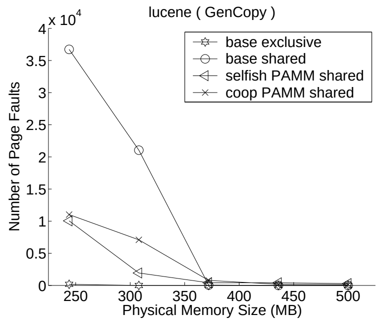
(e) GenCopy running ipsixql



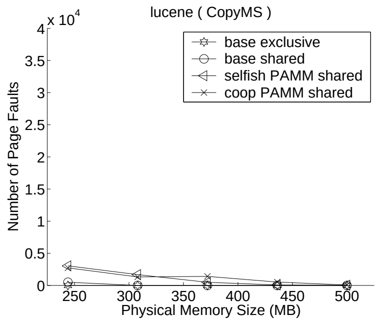
(f) CopyMS running ipsixql



(g) GenMS running lucene



(h) GenCopy running lucene



(i) CopyMS running lucene

Figure 4: Graphs showing the total page faults when varying the amount of physical memory on the machine.

in Figure 3(a). When both processes can collect the heap with GenMS without page faults (e.g., physical memory sizes 376MB and greater), cooperative PAMM and Base Shared provided equally good throughput, with execution times differing by less than a second. At smaller physical memory sizes, cooperative PAMM improves performance by triggering 33% fewer page faults. Similarly, in Figure 3(b), cooperative PAMM not only outperforms Base Shared, but its performance does not suffer from paging at even the smallest physical memory sizes. While Base Shared has a higher throughput at the larger physical memory sizes in a few instances, these differences tend to be small. Unlike Base Shared, our cooperative, resource-based adaptive garbage collection scheme performs well both when memory is plentiful and when memory is scarce.

Cooperative PAMM can also be seen to shine when compared with Base Exclusive. Lacking any competition for physical memory and lacking the means to utilize additional memory, Base Exclusive runs vary little with the physical memory size. These runs therefore usually perform the best at the smallest physical memory sizes, but this does not always hold. In Figures 3(b) and 3(g) cooperative PAMM dominates Base Exclusive, running the benchmarks at least 9% and 15% faster, respectively. In all of our experiments, however, cooperative PAMM outperforms Base Exclusive by at least 10% when memory is most plentiful. In fact, using GenCopy to collect runs of ipsixql (Figure 3(e)) is the only instance where cooperative PAMM does not provide an improvement of at least 20%. So while Base Exclusive *may* provide some improvement when memory is scarce, cooperative PAMM *always* improves throughput when memory is available.

5.3 Cooperative vs. Selfish PAMM

While these results show resource-based garbage collection can improve system throughput, it is instructive to compare results from both cooperative and selfish PAMM. When physical memory is plentiful, the two resource-based garbage collection systems perform similarly. For runs of pseudoJBB at the largest physical memory sizes, execution times for the two systems are always within 3 seconds of one another. The biggest difference occurs when memory pressure is high. Because cooperative PAMM decreases its softbound by a constant step size, it is unable to shrink the heap as quickly as selfish PAMM. On the other hand, cooperative PAMM also begins to shrink all of the applications' heaps as soon as the first application begins to page. While this means that selfish PAMM may perform slightly better when there is a small amount of paging (e.g., Figures 3(c) and 3(f)), it means selfish PAMM suffers up to five times as many page faults as cooperative PAMM at the smallest physical memory sizes. While providing the optimal performance when memory pressure is light, cooperative PAMM also does not suffer from a large number of page faults when memory pressure is high. Combined with its performance when the physical memory is plentiful, this makes the cooperative implementation of PAMM a clear improvement.

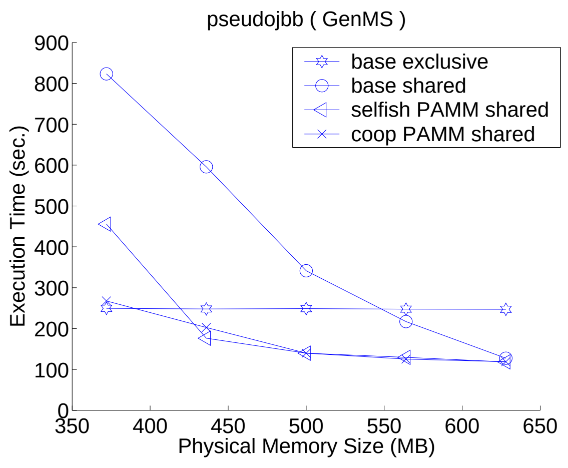
5.4 More concurrent workloads

We conducted experiments of 4 concurrent workloads on a 2-CPU Xeon machine with hyperthreading enabled. Figure 5(a) and 5(b) show the results of pseudojbb and ipsixql when using GenMS. Both are similar to their corresponding 2-workload graphs as Figure 3(a) and 3(d). When memory is plentiful, all three Shared schemes have similar performance – better than Base Exclusive; as page faults rise, cooperative PAMM is the champion until the memory pressure is too high for it to beat Base Exclusive, which keeps zero paging for its much smaller memory demand.

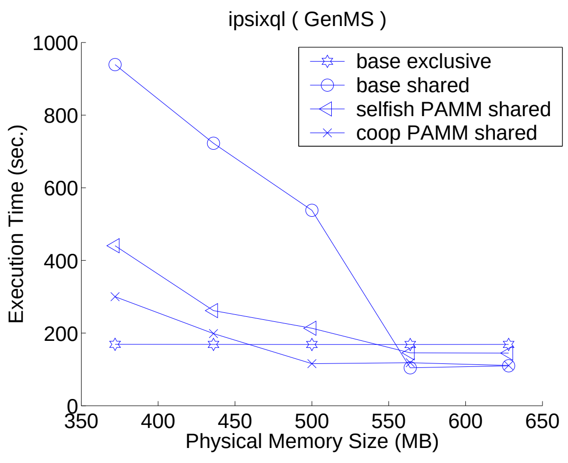
5.5 Fairness and responsiveness

Figures 6 and 7 presents fairness curves for 5 runs of pseudoJBB, where each run consists of 2 simultaneous executions of pseudoJBB. The former figure is for runs using the default memory manager, while the latter figure is for runs using PAMM. Each point on this graph shows the minimum fairness for any given time window or larger; the rightmost point plotted for a run on the graph is the run's throughput – the total time needed to execute the two runs.

Figure 6 shows that the performance of the default memory manager can be highly variable. The first three runs, where execution is virtually serialized, complete in shortest time of any experiments. While this provides very good throughput, this throughput come at a substantial impact on fairness. Figure 7 shows that runs using PAMM are more uniform in execution and exhibit greater levels of fairness. PAMM is clearly doing a good job dividing the memory between the two applications. At the same time, this performance comes with a lower throughput that the first several runs of the default memory manager. Finally the last several runs of the default memory manager show the lowest throughput and fairness. While the default system performs best when applications are executed sequentially, its performs worst when applications are executed in parallel.



(a) GenMS running pseudoJBB



(b) GenMS running ipsixql

Figure 5: Graphs showing the total execution time when varying the amount of physical memory on a 2-CPU Xeon machine with hyperthreading enabled

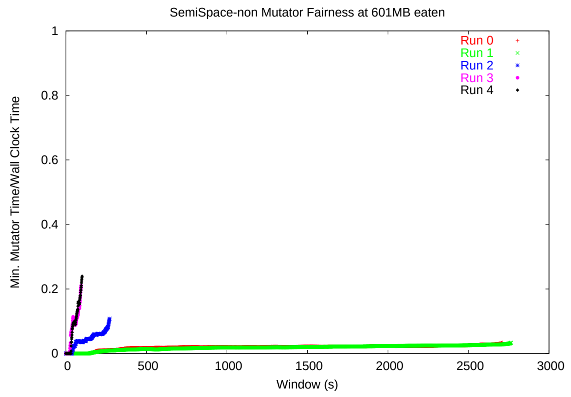


Figure 6: Bounded fairness curves for runs of pseudoJBB using 192MB of physical memory and the default memory manager. Curves to the left and higher are better.

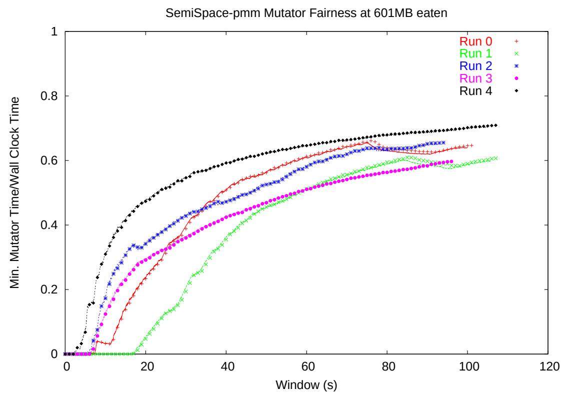
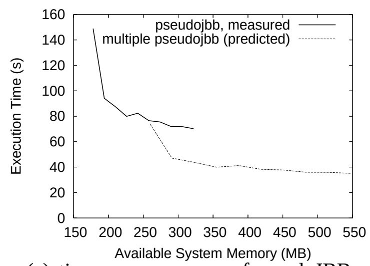
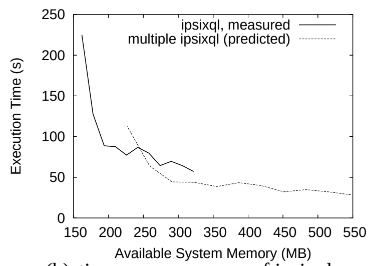


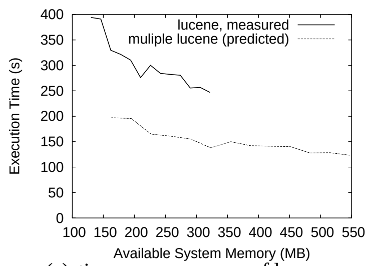
Figure 7: Bounded fairness curves for runs of pseudoJBB using 192MB of physical memory using PAMM. Curves to the left and higher are better.



(a) time-memory curve of pseudoJBB



(b) time-memory curve of ipsixql



(c) time-memory curve of lucene

Figure 8: The time-memory curve using CopyMS and the prediction of parallel execution time. The two curves intersect once the concurrent execution has enough memory to share effectively, and thus achieve better throughput than sequential runs.

5.6 Memory performance prediction

Based on the time-memory curve, we can predict the performance of memory sharing and consequently find the fastest multiprogramming configuration. Solid lines in figure 8(a) to 8(c) show the time-memory curve for the applications $pseudojbb$, $ipsixql$ and $lucene$, when using the PAMM controller with the CopyMS collector. The X axis shows the total amount of memory available for the application and the OS and the y axis shows the execution time.

We now test our conjecture that the time-memory curve predicts the effect of memory sharing. For simplicity, we consider the case where two identical instances running on two processors, for which there are two choices: running them either concurrently with half memory each or sequentially with full memory.

For each point (x, y) of the time-memory curve, we calculate the needed memory and the expected concurrent running time. The memory is $2x - m_{OS}$, because we have two concurrent executions of a program sharing the same OS space. We compare the predicted parallel time, $t^{parallel} = Time(x)$ with the time of running two programs sequentially but having exclusive use of memory, $t^{sequential} = 2 * Time(2x - m_{OS})$. For comparison, we draw $t^{parallel}/2$ by translating each point from (x, y) to $(2x-m_{OS}, y/2)$. The two translated curves are shown in Figure 5.6, with m_{OS} set to 97MB, which accounts for the (minimal) memory reportedly in use without the benchmarks running (38MB), and the smallest amount free memory the operating system would allow (49MB).

For $pseudojbb$, the curve predicts that the choice changes at the breakpoint around 260MB, where the concurrent execution has enough memory to share and to perform better than sequential runs for throughput. Looking at Figure 3(c) one sees that the actual breaking point is between 250MB and 300MB. When comparing the lucene results of Figure 3(i) to the prediction in Figure 8(c) we can see that there is no cross point in either case. Again, the time-memory curve predicts what level of multi-programming would be most efficient.

The prediction is made possible by resource-based adaptive GC, which automatically uses all available memory. This property enables a systematic performance model for a shared environment, where one can quantitatively compare all types of memory sharing, including sequential runs with no memory sharing. In contrast, the conventional scheme is ad hoc and requires experimenting with different heap-size setups when running multiple programs. On-line adaptation is not possible without a resource-based scheme to constantly monitor the amount of available memory and to use exactly that amount.

6 Related work

The prevailing models of memory performance assume that an application does not change its memory demand based on the available memory [10, 15, 19]. Therefore they cannot predict performance of resource-based adaptive memory management in either single or multi-programming environments. We present the time-memory curve as a solution and show experimentally that it predicts the sequential and parallel performance for non-trivial applications.

Resource-based memory management has been studied originally at the level of the virtual machine. Alonso and Appel presented a collector which reduced the heap size when advised that memory pressure was increasing [3]. 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 enabling the JVM to select a heap size that fully utilize physical memory [22, 23]. Instead of changing memory allocation, Hertz et al. developed a paging-aware garbage collector and modified virtual memory manager that cooperated to greatly reduce the paging costs of large heaps [12]. These past schemes require modifications to the virtual machine and, for all but one, the operating system. Program-level techniques do not require these intrusive modifications.

Andreasson et al. used reinforcement learning to improve GC decisions through thousands of iterations. They assigned fixed cost for GC and paging and predicted the running time as a function of these and other parameters [4]. The average performance improvement for SPECjbb2K running on JRockit was 2% with the learning overhead and 6% otherwise. Instead of using a fixed cost and memory size, our recent work adaptively monitored the number of page faults and adjusted the heap size of a program in an exclusive environment [24]. Both of the methods required manual analysis of the program. In addition to the formal modeling, our work complements the past program- and VM-based techniques in two aspects. First, we show that run-time monitoring can be fully automated and can therefore be applied to general programs. More importantly, we develop a cooperative scheme for use of resource-based adaptation in a shared environment.

Many other adaptive schemes have been used for garbage collection. Several recent studies examined adaptation based on the program demand. Buytaert et al. use offline profiling to determine the amount of reachable data as the program runs and generate a listing of program points when collecting the heap will be most favorable. At runtime, they then can then collect the heap when the ratio of reachable to unreachable data is most effective [7]. 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 [11]. Soman et al. used profiling, user annotation, and a modified JVM so a program may select which garbage collector to use at the program loading time [20]. Our work has orthogonal purposes because it automatically adapts to the changing resources at run time.

While heap management adds several new wrinkles, there has long been work on creating virtual memory managers which 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 [18]. Last reuse distance, another recency metric, was used by Jiang and Zhang to avert thrashing [14], by Chen et al. to improve Linux VM [8], and by Zhou et al. to improve multi-programming [25]. All of these techniques try to best allocate physical memory for a fixed subset of the working set, but are of limited benefit when the total working set fits in available memory or when the available memory is too small for the subset. Resource-based memory management, on the other hand, can improve performance when given additional physical memory by reducing the frequency of GC. When the memory is in short supply, it increases the frequency of GC to reduce the the size of the heap. In this paper we have presented a set of techniques that adapt between smaller heap sizes (more frequent garbage collections) and larger heap sizes (an increased chance of paging) dynamically and fairly in a shared environment.

7 Summary

Two major shifts in memory system design are the separation of data and their storage by the virtual memory system and the decoupling of data demand and memory allocation by garbage collection. Resource-based garbage collection connects the program-level allocation and the physical storage by adapting the memory usage with the available resource. The time-memory curve and adaptive GC are effective techniques in avoiding conflicts yet making full uses resources in a dynamically shared environment for garbage-collected programs. The time-memory curve has further implications for the efficacy of multi-programming as opposed to executing programs in series.

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.

This research is supported by the National Science Foundation (Contract No. CNS-0509270, CCR-0238176), IBM CAS Fellowship, two grants from Microsoft Research, and an equipment donation from IBM. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of above named organizations.

References

[1] 20 newsgroups. Available at http://people.csail.mit.edu/jrennie/20Newsgroups/.

[2] lucene. Available at http://lucene.apache.org/java/docs/.

[3] R. Alonso and A. W. Appel. An advisor for flexible working sets. In *Proceedings of the 1990 Joint International Conference on Measurement and Modeling of Computer Systems*, pages 153–162, Boulder, CO, May 1990.

[4] 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.

[5] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *Thirtieth Annual ACM Symposium on Principles of Programming Languages*, New Orleans, LA, Jan. 2003.

[6] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, Sept. 1988.

[7] 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.

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

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

[10] P. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1), January 1980.

[11] 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.

[12] 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.

[13] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications*, volume 39(11), pages 69–80, Vancouver, BC, Canada, Oct. 2004.

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

[15] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[16] N. Sachindran and J. E. B. Moss. Mark-Copy: Fast copying GC with less space overhead. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications*, volume 38(11), pages 326–343, Anaheim, CA, Oct. 2003.

[17] N. Sachindran, J. E. B. Moss, and E. D. Berger. MC^2: High performance garbage collection for memory-constrained enviornments. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Systems, Programming Languages, and Applications*.

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

[19] A. J. Smith. On the effectiveness of set associative page mapping and its applications in main memory management. In *Proceedings of the 2nd International Conference on Software Engineering*, 1976.

[20] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *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, E. D. Berger, S. F. Kaplan, and J. E. Moss. CRAMM: virtual memory support for garbage-collected applications. In *Proceedings of OSDI*, 2006.

[23] 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.

[24] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara. Program-level adaptive memory management. In *Proceedings of the International Symposium on Memory Management*, Ottawa, Canada, June 2006.

[25] 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.