Autonomic Heap Sizing: Taking Real Memory Into Account

Ting Yang tingy@cs.umass.edu
Emery D. Berger emery@cs.umass.edu
Matthew H. Hertz hertz@cs.umass.edu
Scott F. Kaplan† sfkaplan@cs.amherst.edu
J. Eliot B. Moss moss@cs.umass.edu

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

†Department of Computer Science
Amherst College
Amherst, MA 01002-5000

ABSTRACT

The selection of heap size has an enormous impact on the performance of applications that use garbage collection. A heap that barely meets the application's minimum requirements will result in excessive garbage collection overhead, while a heap that exceeds physical memory will cause paging. Choosing the best heap size *a priori* is impossible in multiprogrammed environments, where physical memory allocated to each process constantly changes. This paper presents an autonomic heap-sizing algorithm that one can apply to different underlying garbage collectors with only modest modifications. It relies on a combination of analytical models and detailed information from the virtual memory manager. The analytical models characterize the relationship between collection algorithm, heap size, and footprint. The virtual memory manager tracks recent reference behavior, and reports the current footprint and allocation to the collector. The garbage collector then uses those values as inputs to its model to compute a heap size that maximizes throughput while minimizing paging. We show that by using our adaptive heap sizing algorithm, we can reduce running time over fixed-sized heaps by as much as 90%.

1. INTRODUCTION

Java and C# have helped to make garbage collection (GC) readily available to programmers working on a wide variety of development projects. While GC provides many useful advantages to its users, it also carries a potential liability: page swapping. When collection occurs, the process rapidly traverses nearly all of its pages in a staggering display of poor locality. If those pages are not cached, garbage collection will cause extensive page swapping. Since disks are 5 to 6 orders of magnitude slower than RAM, even modest amounts of page swapping can ruin application performance. It is therefore important that all of the process's pages—its *footprint*—be cached to avoid page swapping overhead.

The footprint of a garbage-collected process is largely determined by one parameter: its *heap size*. A sufficiently small heap size reduces the footprint so that no paging occurs during garbage collection. However, a heap size that is too small causes frequent collections. A process that is collecting too often is not making progress on its intended task and is harming overall system performance.

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.
Copyright 2001 ACM 0-89791-88-6/97/05 ...$5.00.

Ideally, the user would choose the *largest* heap size for which the entire footprint is cached. Such a heap size would trigger garbage collection just often enough to prevent the footprint from expanding beyond the capacity of main memory. The CPU time consumed by collection would be reduced as far as possible without incurring the overhead of page swapping.

Unfortunately, from the standpoint of a single process, the capacity of main memory is not constant. In a multiprogrammed environment, the operating system's virtual memory manager (VMM) must dynamically allocate main memory to each process and to the file system cache. Therefore, the amount of space allocated to one process will change over time in response to *memory pressure*—the demand for main memory space exhibited by the current workload. Even in systems with large main memories, uses of even larger file systems will bring about memory pressure. Disk accesses, whether caused by virtual memory paging or explicit I/O requests, slow system performance equally.

Currently, the user of a garbage-collected application must select a heap size when the process is started. That heap size will not change for the duration of the execution. Even if the user has sufficient information about the state of the system to choose a good initial heap size, the memory pressure may change during execution and cause the choice to become a poor one. Since memory pressure changes dynamically with the system's workload, the heap size of a garbage-collected application should also change in response.

Contributions. We present an adaptive heap-sizing algorithm. It relies on the virtual memory system to provide periodically the current main memory allocation. It then selects a heap size that corresponds to a footprint that just fits the given allocation. This heap size does not induce page swapping at collection time, but fully utilizes the allocated main memory space to reduce CPU time consumed in collection.

In order to map each possible heap size to its footprint, our algorithm relies on an analytic model of the garbage collection algorithm itself. This model uses measurements of the footprint that VMM provides, and calculates the relationship between heap size and footprint experienced thus far in the execution. It then uses this relationship, along with its current allocation size, to select a heap size that will yield an appropriate footprint. We have developed models for both the *semi-space* and *Appel* garbage collectors, and we show that these models generate accurate predictions.

We also present the design for a VMM that can gather the reference distribution data necessary to calculate the current footprint and provide it to the model. This VMM tracks references only to less recently used pages, and thus does not interfere with the vast majority of references that are made to more recently used pages. The VMM adjusts dynamically and online the number of recently used pages whose ref-

erences the VMM does not track, so that the total overhead does not exceed a threshold. Thus, the VMM can gather reference distribution information that is sufficient for our predictive models while adding only 1% to the total running time.

In exchange for this 1% overhead in the VMM, our algorithm dynamically selects a heap size on-line, reducing garbage collection time and nearly eliminating paging. Hence it reduces the total running time by as much as 90%, and typically by 10% to 40%. We show, for a variety of benchmarks, using both *semi-space* and *Appel* collectors, that our algorithm selects good heap sizes for widely varying main memory allocations.

2. RELATED WORK

The problem of heap size selection has received surprisingly little attention, despite its enormous potential impact on application performance. We know of only three papers on the topic. We discuss these, and then turn to existing interfaces to virtual memory managers.

2.1 Heap Sizing

Kim and Hsu use the SPECjvm98 benchmarks in examining the paging behavior of garbage collection [11]. They execute each program with a variety of heap sizes on a system with 32MB of RAM. They observe that performance suffers when the heap does not fit in real memory, and when the heap is larger than real memory it is often better to grow the heap than to collect. Kim and Hsu conclude that there is an optimal heap size for each program for a given real memory. While this may be true, selecting optimal heap sizes *a priori* does not work in the context of multiprogrammed systems where the amount of available memory changes dynamically.

The most similar work to our own is by Alonso and Appel, who also exploit information from the virtual memory manager to adjust heap size [1]. Their garbage collector periodically queries the virtual memory manager to find the current amount of available memory, and then adjusts heap size in response. Our work differs from theirs in several key respects. While their approach can also shrink the heap to avoid paging when memory pressure is high, they do not address the problem of expanding heaps when memory pressure is low. Such heap expansion is crucial in order to reduce the cost of frequent garbage collections. Further, they rely on standard interfaces to virtual memory information, which provides at best a coarse estimate of memory pressure. Our virtual memory management algorithm captures detailed reference information that allows us to calculate the appropriate heap size given available memory.

Brecht et al. adapt Alonso and Appel's approach to control heap growth, but rather than interact with the virtual memory manager, they propose ad hoc rules for two given memory sizes [7]. These memory sizes cannot change, that is, this technique works only if the application is the only program in the system and the user provides the right memory size. Also, their study relied on the Boehm-Weiser mark-sweep collector [6], which can grow its heap but cannot shrink it.

2.2 Virtual Memory Interfaces

Systems typically offer a way for an application to communicate detailed information to the virtual memory manager, but expose very little information in the other direction. Many UNIX and UNIX-like systems support the `madvise` system call, by which applications may communicate detailed information about their reference behavior to the virtual memory manager. An application can indicate that a range of pages will be referenced in a sequential, random, or "normal" manner, will or will not be used soon, or contains no data. No standard dictates how a VMM should respond to these hints.

We know of no systems that expose more detailed information about an application's virtual memory behavior beyond memory residency. The `mincore` system call takes as input a range of memory addresses, and returns an array where each entry is 1 if and only if the corresponding page is resident ("in core"). In the work reported here we use an even simpler interface: the VMM conveys to the program two values: the amount of memory the application needs in order to avoid significant paging (derived from the application's recent reference behavior), and the amount of memory it has available at the present time. The application's memory management code (garbage collector) uses this information to adjust the heap size accordingly.

3. GC PAGING BEHAVIOR ANALYSIS

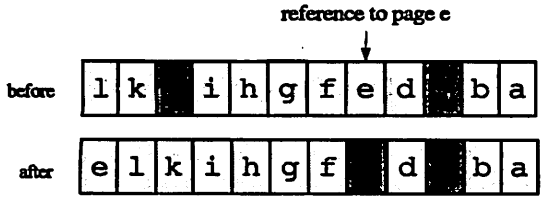
To build robust mechanisms for controlling paging behavior of garbage collected applications it is important first to understand those paging behaviors. Consequently, we studied those behaviors by collecting and analyzing memory reference traces for a set a benchmark programs, when executed under each of several collectors, for each of a number of heap sizes. The goal was to reveal, for each collector, the regularities in the reference patterns and the relationship between heap size and footprint.

Methodology Overview: We used an instrumented version of Dynamic SimpleScalar (DSS) [8] to generate memory reference traces. We pre-processed these with the SAD reference trace reduction algorithm [9, 10]. (SAD stands for Safely Allowed Drop, which will make sense when we explain below our extensions to it.) For a given *reduction memory size* of m pages, SAD produces a substantially reduced trace that will trigger the same exact sequence of faults for a simulated memory of at least m pages, managed with least-recently-used (LRU) replacement. SAD drops most references that hit in memories smaller than m, keeping only the few such reference necessary to ensure that the LRU stack order is the same for pages in stack positions m and beyond. We then processed the SAD-reduced traces with an LRU stack simulator to obtain the number of faults for all memory sizes no smaller than m pages.

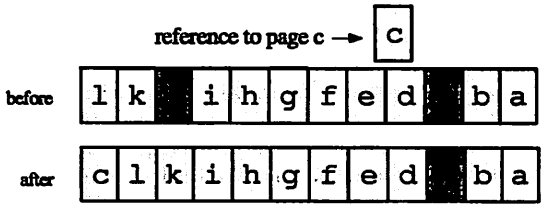
Estimating time: We also obtained a rough estimate of execution time. DSS outputs a count of instructions simulated and a count of memory references (including instruction fetches). We simply charge a fixed number of instructions for each page fault to estimate total execution time. We further assume that writing back dirty pages can be done asynchronously so as to interfere minimally with application execution and paging. We ignore other operating system costs, such as application I/O requests. These modeling assumptions are reasonable because we are interested primarily in order-of-magnitude comparative performance estimates, not in precise absolute time estimates. The specific values we used assume that a processor achieves an average throughput of 1×10^9 instructions/sec and that a page fault stalls the application for 5ms $= 5 \times 10^6$ instructions.

SAD and LRU Extensions: Because our garbage collectors make calls to `mmap` (to request demand-zero pages) and `munmap` (to free regions evacuated by GC), we needed to extend the SAD and LRU models to handle these primitives sensibly. Since SAD and LRU both treat the first access to a page not previously seen as a compulsory miss, `mmap` requires no special handling. We do not charge for compulsory misses. Because the program image and initial heap of the Java system are likely to be contiguous on disk, common OS prefetching mechanisms will fetch this data at far lower cost than normal page faults. Furthermore, the size of this data is orthogonal to the chosen heap size, and thus all heap sizes incur the same amount of I/O to read this initial heap and system.[1] Finally, in a multiprogrammed system, this initial data is liked to be shared `mmap` space, and therefore may

[1]There is also little difference in the size of initial data across collectors because the dynamically allocated heap starts empty; the only difference is the collector code and initial data structures.



(a) Touching a page in the LRU stack



(b) Touching a page not in the LRU stack

Figure 1: LRU Stack Handling of Unmapped Pages

already be resident. We do not change for compulsory references to demand-zero pages either, since they incur only a minor page fault—one that does not require a disk access—to allocate and zero a new page.

We do need to handle `munmap` events specially, however. First we describe how to model unmapping for the LRU stack algorithm, and then describe how to extend the SAD trace reduction algorithm accordingly. Consider the diagram in Figure 1(a). The upper configuration illustrates the state of the stack after the sequence of references `a`, `b`, `c`, `d`, `e`, `f`, `g`, `h`, `i`, `j`, `k`, `l`, followed by unmapping of `c` and `j`.

Note that we leave *place holders* in the LRU stack for the unmapped pages. Now suppose the next reference is to page `e`. We bring `e` to the front of the stack, and move the first unmapped page place holder to where `e` was in the stack. Why is this correct? For memories of size 2 or less, it reflects the page-in of `e` and the eviction of `k`. For memories of size 3 through 7, it reflects the need to page in `e`, and that, because there is a free page, there is no need to evict a page. For memories of size 8 or more, it reflects that there will be no page-in or eviction. Note that if the next reference had been to `k` or `l`, we would not move any place holder, and if the next reference had been to `a`, the place holder between `k` and `i` would move down to the position of `a`. When a place holder reaches the old end of the stack (the right as we have drawn it), it may be dropped.

Now consider Figure 1(b), which shows what happens when we reference a page *not* in the LRU stack (a compulsory miss, which may be to a page never before seen, or to a page that was unmapped and then mapped demand-zero). In this case the reference is to page `c`. We push `c` onto the front of the stack, and slide the previously topmost elements to the right, until we consume one place holder (or we reach the end of the stack). This is correct because it requires a page-in for all memory sizes, but requires eviction only for memories of size less than 3, since the third slot is free.

One might be concerned that the place holders can cause the LRU stack structure to grow without bound. However, because of the way compulsory misses are handled (Figure 1(b)), the stack will in fact never contain more elements than the maximum number of pages mapped at one time by the application.

To explain the modifications to SAD, we first provide a more detailed overview of its operation. Given a reduction memory size m, SAD maintains an m-page LRU stack as well as a window of references from the reference trace being reduced. Critically, this window contains references only to those pages that are currently in the m-page LRU stack. Adding the next reference from the source trace to the front of this window triggers one of two cases. The first case applies if the reference does not cause eviction from the LRU stack (i.e., the stack is not full or the reference is to one of the m most recently used pages). For this case, the reference is added to the window. Furthermore, if the window contains *two* previous references to the same page, SAD deletes the *middle* reference, since the absence of that reference does not affect the evicting and fetching of that page from an m-page memory (and hence from any larger memory).

The second case occurs when the reference causes an eviction from the m-page LRU stack. If p is the evicted page, then SAD removes references from the back of the window, emitting these references to the reduced trace file, until no references to p remain in the window. This step preserves the window's property of containing references only to pages that are contained in the m-page LRU stack. At the end of the program run, SAD flushes the remaining contents of the window to the reduced trace file.

An unmapped page will affect SAD only if it is one of the m most recently used pages. If this case occurs, it is adequate to update the LRU stack by dropping the unmapped page and sliding other pages towards the more recently used end of the stack to close up the gap. Since that unmapped page no longer exists in the LRU stack, references to it must be removed from the window. Our modified SAD handles this case as it would an evicted page, emitting references from the back of the window until it contains no more references to the unmapped page.[2]

Application platform: We used Jikes RVM version 2.0.1 [3, 2] built for the PowerPC architecture as our Java platform. We optimized the system images to the highest optimization level and included all normal run-time system components in the images, to avoid run-time compilation of those components. The most cost-effective mode for running Jikes RVM is with its *adaptive* compilation system, which compiles application code first with a quick non-optimizing compiler, and then detects frequently executed ("hot") code and optimizes it at progressively higher levels if it stays hot. Because the adaptive system uses timer-driven sampling to invoke optimization, it is non-deterministic. We desired comparable non-deterministic executions to make our experiments repeatable, so we took compilation logs from a number of runs of each benchmark in the adaptive system, determined the median optimization level for each method, and directed the system to compile each method to that method's median level as soon as the system loaded the method. We call this the *pseudo-adaptive* system, and it indeed achieves the goals of determinism and high similarity to typical adaptive system runs.

Collectors: We considered three collectors: mark-sweep (MS), semi-space copying collection (SS), and Appel-style generational copying collection (Appel) [4]. MS is one of the original "Watson" collectors written at IBM. It uses segregated free lists and separate spaces and GC triggers for small versus large objects (where "large" means more than 2KB). MS allows allocation until either the small or large space fills, and then it does marking and sweeping of both heaps, returning freed space to the segregated lists. SS and Appel come from the Garbage Collector Toolkit (GCTk), developed at The University

[2]This approach also maintains SAD's guarantee that the window never holds more than $2m + 1$ entries.

of Massachusetts Amherst and contributed to the Jikes RVM open source repository. They do not have a separate space for large objects. SS is a straightforward copying collector that triggers collection when a semi-space (half of the heap) fills, copying reachable objects to the other semi-space. Appel adds a nursery, where it allocates all new objects. Nursery collection copies survivors to the current old-generation semi-space. If the space remaining is too small, it then does an old-generation semi-space collection. In any case, the new nursery size is half the total heap size allowed, minus the space used in the old generation. Both SS and Appel allocate linearly in their allocation area.

Benchmarks: We use a representative selection of programs from SPECjvm98. We also use `ipsixql`, an XML database program, and `pseudojbb`, which is the SPECjbb2000 benchmark modified to perform a fixed number of iterations (thus making time and GC comparisons more meaningful). We ran all these on their "large" (a size of 100) inputs.

3.1 Results and Analysis

We consider the results for `jack` and `javac` under the SS collector. The results for the other benchmarks are strongly similar, and so we present these two benchmarks as representative of the others. Figure 2 shows the number of page faults for varying main memory allocations. Each curve in each graph comes from one simulation run of the benchmark in question at a particular main memory allocation. Note that the vertical scales are *logarithmic*. Notice that the final drop in each curve happens in order of increasing heap size, i.e., the smallest heap size drops to zero page faults at the smallest allocation.

We notice that each curve has three regions. At the smallest memory sizes, we see extremely high amounts of page swapping. Curiously, larger *heap* sizes perform better for these small memory sizes! This happens because most of the paging occurs during collection, and a larger heap size yields fewer collections, and thus less page swapping.

The second region of each curve is a broad, flat region representing substantial page swapping. For a range of main memory allocations, the program repeatedly allocates in the heap until the heap is full, and the collector then walks over most of the heap, copying reachable objects. Both steps are similar to looping over a large array, and require an allocation equal to a semi-space to avoid paging.[3]

Finally, the third region of each curve is a sharp drop in faults that occurs once the allocation is large enough to capture the "looping" behavior. The final drop occurs at an allocation that is near to half of the heap size plus a constant (about 30MB for `jack`). This regularity suggests that there is a base amount of memory needed for the Jikes RVM system and the application code, plus additional space for a semi-space from the heap.

We further notice that for most memory sizes, GC faults dominate mutator (application) faults. Furthermore, mutator faults have a component that depends on heap size. This dependence results from the mutator's allocation of objects in the heap between collections.

The behavior of MS strongly resembles the behavior of SS, as shown in Figure 3. The final drop in the curves tends to be at the heap size plus a constant, which is logical in that MS allocates to its heap size, and then collects. MS shows other plateaus, which we suspect have to do with their being some locality in each free list, but the page swapping experienced on even the lowest plateau gives a substantial increase in program running time. It is important to select a heap size whose final drop-off is contained by the current main memory allocation.

The curves for Appel (Figure 4) are also more complex than those for SS, but show the same pattern of a final drop in page faulting at 1/2 the heap size plus a constant.

[3]The separate graphs for faults during GC and faults during mutator execution support this conclusion.

3.2 Proposed Heap Footprint Model

These results lead us to propose that the minimum real memory R required to run an application at heap size h without substantial paging is approximately $a \cdot h + b$, where a is a constant that depends on the GC algorithm (1 for MS and 0.5 for SS and Appel) and b depends partly on Jikes RVM and partly on the application itself. The intuition behind the formula is this: an application repeatedly fills its available heap ($1/2 \cdot h$ for Appel and SS; h for MS), and then, during a full heap collection, copies out of that heap the portion that is live (b).

In sum, we suggest that required real memory is a linear function of heap size. We tested this hypothesis using results derived from those already presented. In particular, suppose we choose a threshold value t, and we desire that the estimated paging cost not exceed t times the application's running time with no paging. For a given value of t, we can plot the minimum main memory allocation required for each of a range of heap sizes such that the paging overhead not exceed t.

Figure 5 shows, for `jack` and `javac` and the three collectors, plots of the main memory allocation necessary at varying heap sizes such that paging remains within a range of thresholds. What we see is that the linear model is excellent for MS and SS, and still good for Appel, across a large range of heap sizes and thresholds. For Appel, beyond a certain heap size there are nursery collections but no full heap collections. At that heap size, there is a "jump" in the curve, but on each side of this heap size there are two distinct regimes that are both linear.

For some applications, our linear model does not hold as well. Figure 6 shows results for `compress` under Appel and SS. For smaller threshold values the linear relationship is still strong, modulo the shift from some full collections to none in Appel. While we note that larger threshold values ultimately give substantially larger departures from linearity, users are most likely to choose small values for t in an attempt nearly to eliminate page swapping. Only under extreme memory pressure would a larger value of t be desirable. The linear model appears to hold well enough for smaller t to consider using it to drive an adaptive heap-sizing mechanism.

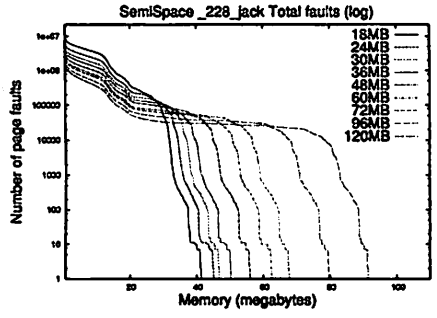
4. DESIGN AND IMPLEMENTATION

The model that correlates heap size and memory footprint, described in Section 3.2, allows one to take as input the current footprint of the application and the current allocation to the process, and then to select a good heap size. To implement this algorithm, we therefore modified two garbage collectors as well as the underlying virtual memory manager (VMM). Specifically, we changed the VMM to collect information sufficient to calculate the footprint, and changed the garbage collectors to adjust the heap size on the fly. Furthermore, we altered the VMM to communicate to the collectors the information necessary to perform the heap size calculation.

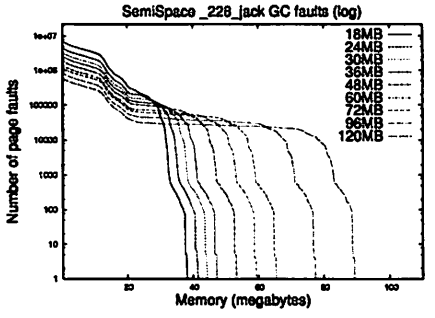
We implemented the modified garbage collectors within the Jikes RVM [3, 2] Java system, which we ran on Dynamic SimpleScalar [8]. This is much the same setup we used to generate the traces we discussed in Section 3.2. However, rather than generating traces, we used a differently extended version of DSS, which models an operating system's VMM. We now proceed to describe this VMM emulator and the modifications to the collectors.

4.1 Emulating a Virtual Memory Manager

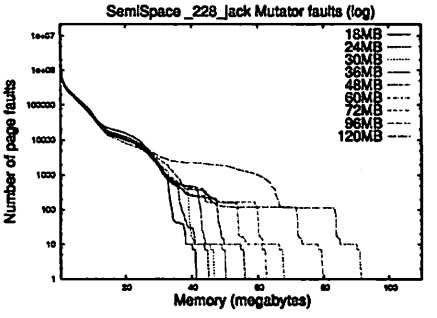
DSS is an instruction-level CPU simulator that emulates the execution of a process under PPC Linux. Since the process requires the services of the underlying operating system, DSS emulates those services, but does so without implementing a full OS kernel. We enhanced the



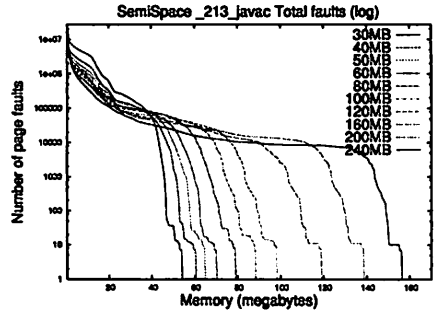
(a) SS total faults for `jack`



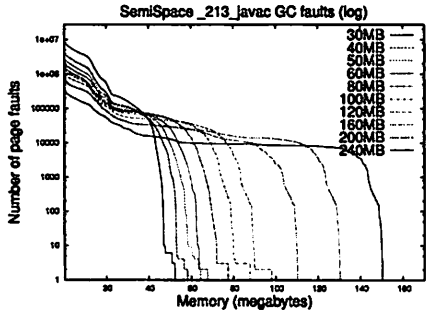
(b) SS GC faults for `jack`



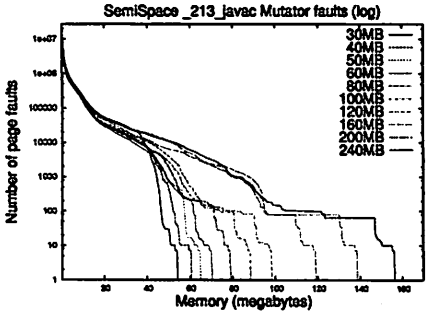
(c) SS mutator faults for `jack`



(d) SS total faults for `javac`

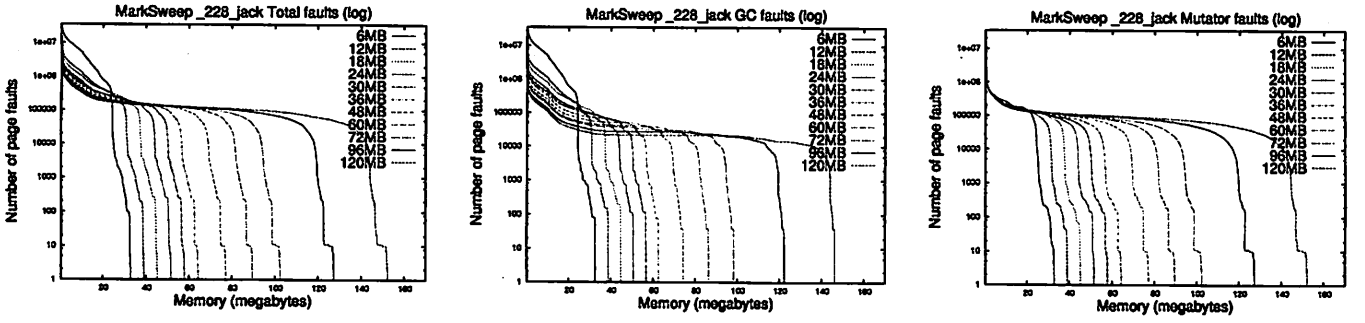


(e) SS GC faults for `javac`

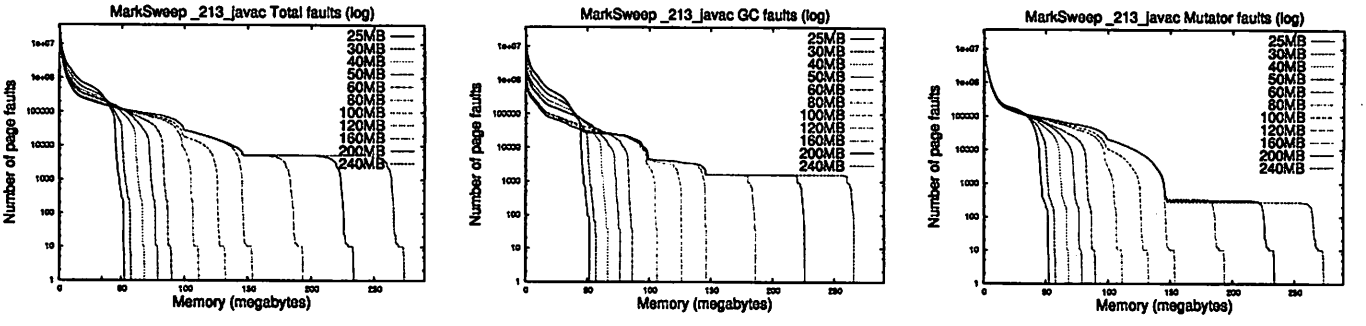


(f) SS mutator faults for `javac`

Figure 2: SS: Faults and estimated time according to memory size and heap size

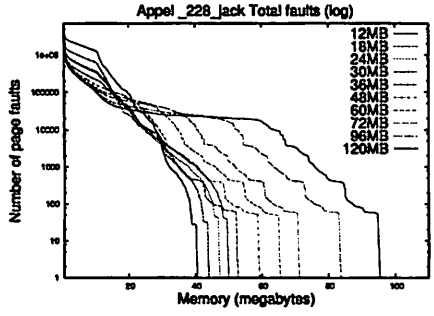


(a) MS total faults for `jack` (b) MS GC faults for `jack` (c) MS mutator faults for `jack`

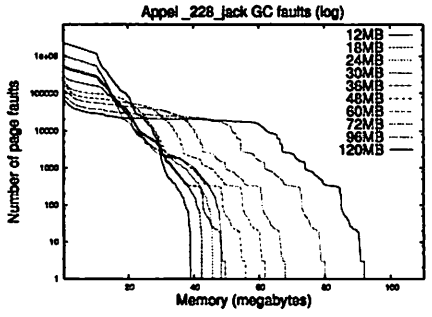


(d) MS total faults for `javac` (e) MS GC faults for `javac` (f) MS mutator faults for `javac`

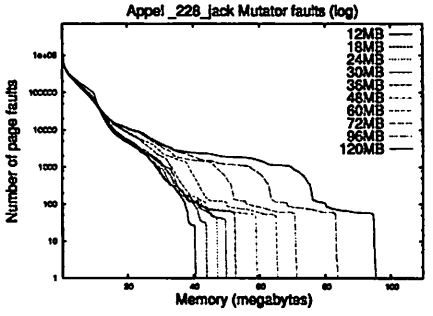
Figure 3: MS: Faults and estimated time according to memory size and heap size



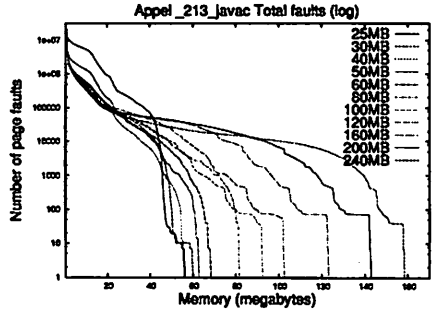
(a) Appel total faults for `jack`



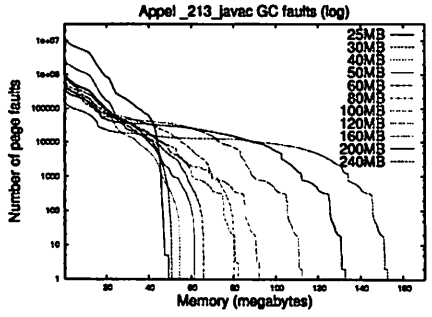
(b) Appel GC faults for `jack`



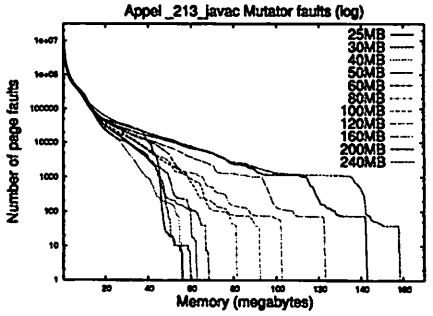
(c) Appel mutator faults for `jack`



(d) Appel total faults for `javac`

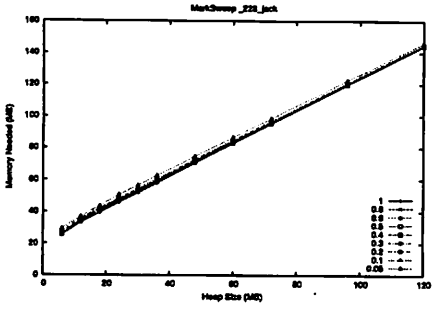


(e) Appel GC faults for `javac`

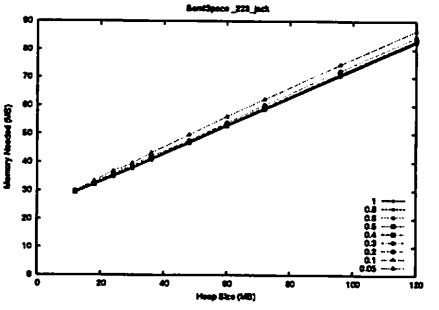


(f) Appel mutator faults for `javac`

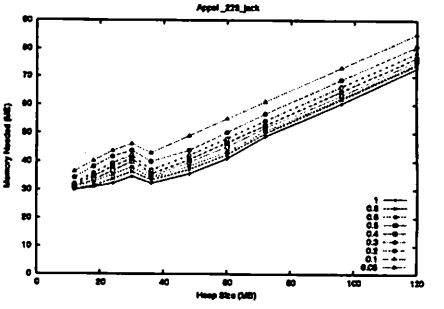
Figure 4: Appel: Faults and estimated time according to memory size and heap size



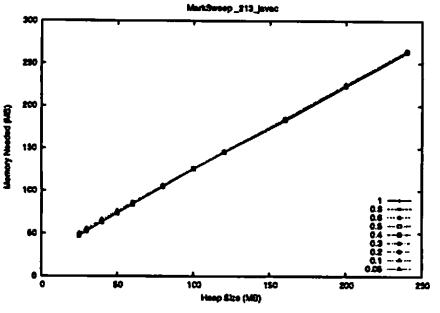
(a) Memory needed for `jack` under MS



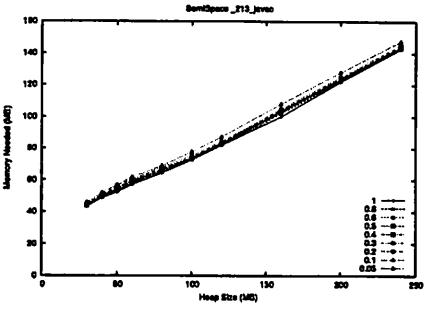
(b) Memory needed for `jack` under SS



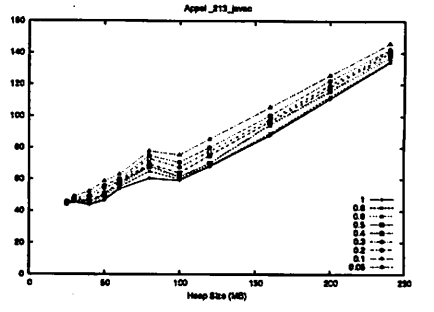
(c) Memory needed for `jack` under Appel



(d) Memory needed for `javac` under MS

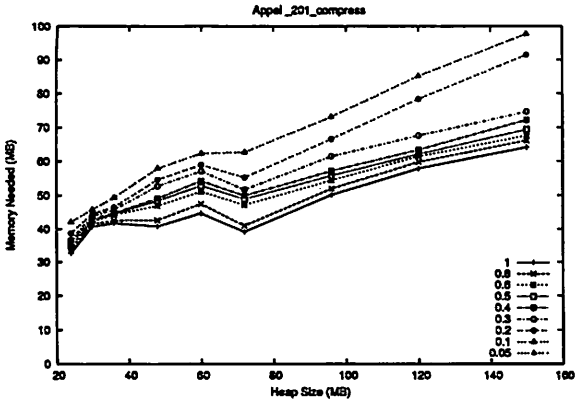


(e) Memory needed for `javac` under SS

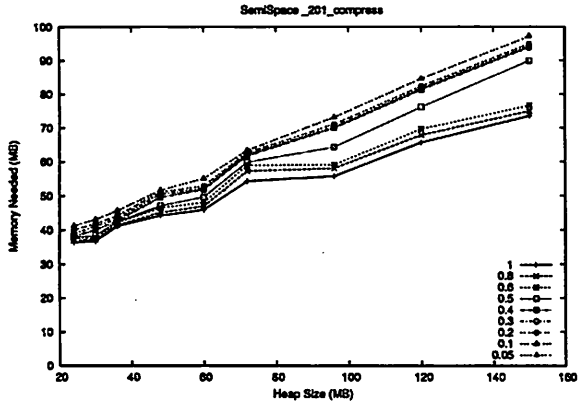


(f) Memory needed for `javac` under Appel

Figure 5: (Real) memory required across range of heap sizes to obtain given paging overhead



(a) Memory needed for `compress` under Appel



(b) Memory needed for `ipsixql` under Appel

Figure 6: (Real) memory required to obtain given paging overhead

emulation of the VMM provided by DSS so that it more realistically modeled a real VMM. Since our algorithm relies on a VMM that communicates both the current allocation and the current footprint to the garbage collector, it is critical that the emulated VMM be sufficiently realistic to approximate the overhead that our methods would impose on a real VMM.

Information collection vs. overhead. The primary responsibility of a VMM is to implement a page replacement policy. It is important that the replacement policy evict to disk pages that will not be used soon; otherwise, performance will suffer due to heavy page swapping. To select such pages, the VMM must keep some amount of information about past memory references in order to predict future reference patterns. However, it is also important that the VMM impose minimal run-time overhead in obtaining this information.

Consequently, real VMMs do not record information about the vast majority of memory references. Instead, they use one or both of the following methods to collect sufficient information with low overhead:

1. **Hardware reference bits:** When the program references a page, the CPU automatically sets a bit associated with that page. Only the VMM can clear the bit, and so it can periodically check it to determine whether the page has been referenced recently. The CLOCK algorithm, which approximates the common *least recently used (LRU)* algorithm, relies on reference bits.
2. **Page protection:** The VMM can remove all access permissions to a page. When that page is next referenced, it will cause a minor page fault (i.e., a fault, but one that does not require disk access to service), thus allowing the VMM to record information when the reference happens. The *Segmented Queue (SEGQ)* technique [5], which also approximates LRU, uses page protections.

A low cost replacement policy. We combine these methods in our emulated VMM. Specifically, we use a SEGQ structure; that is, main memory is divided into two segments where the more recently used pages are placed in the first segment—a *hot set* of pages—while less recently used pages are in the second segment—the *cold set*. When a new page is faulted into main memory, it is placed in the first (hot) segment. If that segment is full, one page is moved into the second segment. If the second segment is full, one page is evicted to disk, thus becoming part of the *evicted set*.

We use the CLOCK algorithm for the hot set. This use of hardware reference bits allows pages to be moved into the cold set in an order that is close to true LRU order. Our model keeps (in software) 8 reference bits. As the CLOCK passes a particular page, we shift its byte of reference bits left by one position and *or* the hardware *referenced* bit into the low position of the byte. The rightmost *one* bit of the reference bits determines the relative age of the page. When we need to evict a hot set page to the cold set, we choose the page of oldest age that comes first after the current CLOCK pointer location.

We apply page protection to pages in the cold set, and store the pages in order of their eviction from the hot set. If the program references a page in the cold set, the VMM restores the page's permissions and moves it to the hot set, potentially forcing some other page out of the hot set and into the cold set. Thus, the cold set behaves like a normal LRU queue.

We modified DSS to emulate both hardware reference bits and protected pages. Our emulated VMM uses these capabilities to implement our CLOCK/LRU SEGQ policy. For a given main memory size, it records the number of minor page faults on protected pages and the number of major page faults on non-resident pages. We can later ascribe service times for minor and major fault handling and thus determine the running time spent in the VMM.

Handling unmapping. As was the case for the SAD and LRU algorithms, our VMM emulation needs to deal with unmapping of pages. The cold and evicted sets work essentially as one large LRU queue, so we handle unmapped pages for those portions as we did for the LRU stack algorithm. As for the hot set, suppose an *unmap* operations causes k pages to be unmapped in the hot set. Our strategy is to shrink the hot set by k pages and put k place holders at the head of the cold set. We then allow future faults from the cold or evicted set to grow the hot set back to its target size.

4.2 Virtual Memory Footprint Calculations

Existing real VMMs lack capabilities critical for supporting our heap sizing algorithm. Specifically, they do not gather sufficient information to calculate the footprint of a process, and they lack a sufficient interface for interacting with our modified garbage collectors. We describe the modifications required to a VMM—modifications that we applied to our emulated VMM—to add these capabilities.

We have modified our VMM to measure the current footprint of

a process, where the *footprint* is defined as the *smallest allocation whose page faulting will increase the total running time by more than a fraction t over the non-paging running time.*[4] When $t = 0$, the corresponding allocation may be wasting space to cache pages that receive very little use. When t is small but non-zero, the corresponding allocation may be substantially smaller in comparison, and yet still yield only trivial amounts of page swapping, so we think non-zero thresholds lead to a more useful definition of *footprint*.

LRU histograms. In order to calculate this footprint, the VMM records an *LRU histogram* [12, 13]. Imagine maintaining an LRU queue, where the positions are numbered starting at 1. Also imagine maintaining a count of the references to pages found at each queue position—that is, for each reference to a page found at position i, we increment a count $H[i]$. This histogram allows the VMM to calculate the number of page faults that would occur with each possible allocation to the process. The VMM finds the footprint by finding the allocation size where the number of faults is just below the number that would cause the running time to exceed the threshold t.

Updating a true LRU queue would impose too much overhead in a real VMM. Instead, our VMM uses the SEGQ structure described in Section 4.1 that approximates LRU at low cost. Under SEGQ, we do not collect histogram information on references to pages in the hot set. Instead, we maintain histogram counts only for references to pages in the cold and evicted sets. Such references incur a minor or major fault, respectively, and thus give the VMM an opportunity to increment the appropriate histogram entry. Since the hot set is much smaller than the footprint, the missing histogram information on the hot set does not harm the footprint calculation.

In order to avoid large space overheads, the VMM also does not maintain one histogram entry per queue position. Instead, we group positions together into bins. Specifically, we use one bin for each 64 pages (256KB given our page size of 4KB). This granularity is fine enough to provide a sufficiently accurate footprint measurement while reducing the space overhead substantially.

Mutator vs. collector referencing. The mutator and garbage collector are likely to exhibit drastically different reference behaviors. Furthermore, when the new heap size is chosen, the reference pattern of the garbage collector will change accordingly, while the reference pattern of the mutator will likely remain similar (in general not exactly the same, since the collector may have moved objects the mutator will reference).

Therefore, the VMM relies on notification from the garbage collector when collection begins and when it ends. One histogram records the mutator's reference pattern, and another histogram records the collector's. When the heap size changes, we clear the collector's histogram, since the previous histogram data no longer provides a meaningful projection of future memory needs.

When the VMM calculates the footprint of a process, it combines the counts from both histograms, thus incorporating the page faulting behavior of both phases.

Unmapping pages. A garbage collector may elect to *unmap* a virtual page, thereby removing it from use. As we discussed previously, we use place holders to model unmapped pages. They are crucial not only in determining the correct number of page faults for each memory size, but also in maintaining the histograms correctly, since the histograms indicate the number of faults one would experience at various memory sizes.

Histogram decay. Programs exhibit *phase behavior*: during a phase, the reference pattern is constant, but when one phase ends and another begins, the reference pattern may change dramatically. Therefore, the histograms must reflect the referencing behavior from the current phase. During a phase, the histogram should continue to accumulate. When a phase change occurs, the old histogram values should be decayed rapidly so that the new reference pattern will emerge.

Therefore, the VMM periodically applies an *exponential decay* to the histogram. Specifically, it multiplies each histogram entry by a decay factor $\alpha = \frac{63}{64}$, ensuring that older histogram data has diminishing influence on the footprint calculation. Previous research has shown that the decay factor is not a sensitive parameter when using LRU histograms to guide adaptive caching strategies [12, 13].

To ensure that the VMM applies decay more rapidly in response to a phase change, we must identify when phase changes occur. Phases are *memory size relative*: a phase change for a hardware cache is not a phase change for a main memory. Therefore, the VMM must respond to referencing behavior near the main memory allocation for the process. Rapid referencing of pages that substantially affect page replacement for the current allocation indicate that a phase change relative to that allocation size is occurring [12, 13].

The VMM therefore maintains a *virtual memory clock* (this is quite distinct from, and should not be confused with the clock of the CLOCK algorithm). A reference to a page in the evicted set advances the clock by 1 unit. A reference to a page in the cold set, whose position in the SEGQ system is i, advances the clock by $f(i)$. If the hot set contains h pages, and the cold set contains c pages, then $h < i \leq h + c$ and $f(i) = \frac{i-h}{c}$.[5] The contribution of the reference to the clock's advancement increases linearly from 0 to 1 as the position nears the end of the cold set, thus causing references to pages that are near to eviction to advance the clock more rapidly.

Once the VMM clock advances $\frac{M}{16}$ units for an M-page allocation, the VMM decays the histogram. The larger the memory, the longer the decay period, since one must reference a larger number of previously cold or evicted pages to constitute a phase change.

Hot set size management. A typical VMM uses a large hot set to avoid minor faults. The cold set is used as a "last chance" for pages to be re-referenced before being evicted to disk. In our case, though, we want to maximize the useful information (LRU histogram) that we collect, so we want the hot set to be as *small* as possible, without causing undue overhead from minor faults. We thus set a target *minor fault overhead*, stated as a fraction of application running time, say 1% (a typical value we used). Periodically (described below) we consider the overhead in the recent past. We calculate this as the (simulated) time spent on minor faults since the last time we checked, divided by the total time since the last time we checked. For "time" we use the number of instructions simulated, and assume an approximate execution rate of 10^9 instructions/sec. We charge 2000 instructions (equivalent to $2\mu s$) per minor fault. If the overhead exceeds 1.5%, we increase the hot set size; if it is less than 0.5%, we decrease it (details in a moment). This simple adaptive mechanism worked quite well to keep the overhead within bounds, and the 1% value provided information good enough for the rest of our mechanisms to work.

[4] *Footprint* has sometimes been used to mean the total number of unique pages used by a process, and sometimes the memory size at which no page faulting occurs. Our definition is taken from this second meaning. We choose not to refer to it as a *working set* because that term has a larger number of poorly defined meanings.

[5] If the cold set is large, the high frequency of references at lower queue positions may advance the clock too rapidly. Therefore, for a total allocation of M pages, we define $c' = max(c, \frac{M}{2})$, $h' = min(h, \frac{M}{2})$, and $f(i) = \frac{i-h'}{c'}$.

How do we add or remove pages from the hot set? Our technique for growing the hot set by k pages is to move into the hot set the k hottest pages of the cold set. To shrink the hot set to a target size, we run the CLOCK algorithm to evict pages from the hot set, but *without* updating the reference bits used by the CLOCK algorithm. In this way the oldest pages in the hot set (insofar as reference bits can tell us age) end up at the head of cold set, with the most recently used nearer the front (i.e., in proper age order).

How do we trigger consideration of hot set size adjustment? For the case where we might want to grow the hot set, we count what we call *hot set ticks*. Given the LRU stack position numbering given above, we associated a weight with each queue position from $h+1$ through $h+c$, such that position $h+1$ has weight 1 and $h+c+1$ has weight 0, i.e., the weight $w = (h+c+1-i)/c$. (This weighting works oppositely to that used for the VMM clock that drives in histogram aging.) For each minor fault that hits in the cold set, we increment the hot set tick count by the weight of the position of the fault. When the tick count exceeds 1/4 the size of the hot set (representing somewhat more than 25% turnover of the hot set), we trigger a size adjustment test. Note that we count faults near the hot set boundary more than ones far from it. The reasoning here is that if we have a high overhead that we can fix with reasonable hot set growth, we will find it more quickly; conversely, if we have many faults from the cold end of the cold set, we may be encountering a phase change in the application and should be careful not to adjust the hot set size too eagerly.

To handle the case where we should consider shrinking the hot set, we consider the passage of (simulated) real time. If, when we handle a fault, we find that we have not considered an adjustment within τ seconds, we trigger consideration. We use a value of 16×10^6 instructions, corresponding to $\tau = 16$ms.

When we want to grow the hot set, how do we compute a new size? Using the current overhead, we determine the number of faults by which we exceeded our target overhead since the last time we considered adjusting the hot set size. We multiply this times the average hot-tick weight of minor faults since that time, namely *hot ticks / minor faults*; we call the resulting number N:

$$W = \textit{hot ticks}/\textit{minor faults}$$

$$\textit{target faults} = (\Delta t \times 1\%)/2000$$

$$N = W \times (\textit{actual faults} - \textit{target faults})$$

Multiplying by the factor W avoids adjusting too eagerly. Using recent histogram counts for pages at the hot end of the cold set, we add pages to the hot set until we have added ones that account for N minor faults since the last time we considered adjusting the hot set size.

When we want to shrink the hot set, how do we compute a new size? In this case, we do not have histogram information, so we assume that (for changes that are not too big) the number of minor faults changes linearly with the number of pages removed from the hot set. Specifically, we compute a desired fractional change:

$$\textit{fraction} = (\textit{target faults} - \textit{actual faults})/\textit{target faults}$$

Then, to be conservative, we reduce the hot set size by only 20% of this fraction:

$$\textit{reduction} = \textit{hot set size} \times \textit{fraction} \times .20$$

We found this scheme to work very well in practice.

VMM/GC interface. The GC and VMM communicate with system calls. The GC initiates communication at the beginning and ending of each collection. When the VMM receives a system call marking the beginning of a collection, it switches from the mutator to the collector histogram. It returns no information to the GC at that time.

When the VMM receives a system call for the ending of a collection, it performs a number of tasks. First, it calculates the footprint of the process based on the histograms and the threshold t for page faulting. Second, it determines the current main memory allocation to the process. Third, it switches from the collector to the mutator histogram. Finally, it returns to the GC the footprint and allocation values. The GC may use these values to calculate a new heap size such that its footprint will fit into its allocated space.

4.3 Adjusting Heap Size

In Section 3 we described the virtual memory behavior of the MS, SS, and Appel collectors in Jikes RVM. We now describe how we modified the SS and Appel collectors so that they modify their heap size in response to available real memory and the application's measured footprint. (Note that MS, unless augmented with compaction, cannot readily shrink its heap, so we did not modify it and drop it from further consideration.) We consider first the case where Jikes RVM starts with the heap size requested on the command line, and then adjusts the heap size after each GC in response to the current footprint and available memory. This gives us a scheme that at least potentially can adapt to changes in available memory during a run. Next, we augment this scheme with a startup adjustment, taking into account from the beginning of a run how much memory is available at the start. We describe this mechanism for the Appel collector, and at the end describe the (much simpler) version for SS.

Basic adjustment scheme. We adjust the heap size after each GC, so as to derive a new nursery size. First, there are several cases in which we do *not* try to adjust the heap size:

- When we just finished a nursery GC that is triggering a full GC. We wait to adjust until after the full GC.
- On startup, i.e., before there are any GCs. (We describe later our special handling of startup.)
- If the GC was a nursery GC, and the nursery was "small", meaning less than 1/2 of the maximum amount we can allocate (i.e., less than 1/4 of the current total heap size). Footprints from small nursery collections tend to be misleadingly small. We call this constant the *nursery filter factor*, which controls which nursery collections heap size adjustment should ignore.

Supposing none of these case pertain, we then act a little differently after nursery versus full GCs. After a nursery GC, we first compute the survival rate of the just completed GC (bytes copied divided by size of from-space). If this survival rate is greater than any survival rate we have yet seen, we estimate the footprint of the *next* full GC. This estimate is:

$$\textit{current footprint} + 2 \times \textit{survival rate} \times \textit{old space size}$$

where the *old space size* is the size *before* this nursery GC.[6] We call this footprint estimate the *estimated future footprint*, or *eff* for short. If the *eff* is less than available memory, we make no adjustment. The point of this whole calculation is to prevent over-eager growing of the heap after nursery GCs. Nursery GC footprints tend to be smaller than full GC footprints; hence our caution about using them to grow the heap.

If the *eff* is more than available memory, or if we just performed a full heap GC, we adjust the heap size, as we now describe. Our first step is to estimate the slope of the footprint versus heap size curve

[6] The factor $2 \times$ *survival rate* is intended to estimate the volume of old space data referenced and copied. It is optimistic about how densely packed the survivors are in from-space. A more conservative value for the factor would be $1 +$ *survival rate*.

(corresponding to the slope of the lines in Figure 5. In general, we use the footprint and heap size of the two most recent GCs to determine this slope. However, after the first GC we have only one point, so in that case we assume a slope of 2 (for $\Delta heap\ size/\Delta footprint$). Further, if we are considering *growing* the heap, we multiply the slope by 1/2, to be conservative. We call constant the *conservative factor* and use it to control how conservatively we should grow the heap. In Section 5, we provide a sensitivity analysis for the *conservative* and *nursery filter factors*.

Using simple algebra, we compute the target heap size from the slope, current and old footprint, and old heap size. ("Old" means after the previous GC; "current" means after the current GC.) Here is the equation:

$$target\ size = old\ size + slope \times (current\ footprint - old\ footprint)$$

We use that target size, subject to two constraints:

1. We will not grow the heap beyond the maximum that Jikes RVM currently supports (256MB).
2. We will not adjust the heap size if the target is less than that required for "reasonable operation". That amount is the size of old space after the current collection, plus the size of the allocation request that triggered GC, plus 1/8 of the target usable heap size. The *usable* heap size is 1/2 the heap size, so the final addend is 1/16 of the target heap size. It is intended to represent the minimum acceptable nursery size to prevent GC from being called outrageously often.

Finally, we note that our calculation is done in terms of 128 KB blocks, not bytes, and is rounded down, which makes it slightly conservative.

Startup heap size. We found that the heap size adjustment algorithm we gave above work well much of the time, but has difficulty if the initial heap size (given by the user on the Jikes RVM command line) is larger than the footprint. The underlying problem is that the first GC causes a lot of paging, yet we do not adjust the heap size until after that GC. Hence we added a startup adjustment. From the currently available memory (a value supplied by the VMM on request), we compute an maximum acceptable heap size:

$$max\ heap\ size = 2 \times (available - 20\text{MB})$$

If the requested heap size exceeds this maximum, we use the computed maximum in its place. Thereafter we adjust the heap as described above.

Heap size adjustment for SS. SS in fact uses the same adjustment *algorithm* as Appel. The critical difference is that in SS there are no nursery GCs, only full GCs.

5. EXPERIMENTAL EVALUATION

To test our algorithm we ran each benchmark described in Section 3 using the range of heap sizes used in Section 3.2 and a selection of fixed main memory allocation sizes. We used each combination of these parameters with both the standard garbage collectors (which use a static heap size) and our dynamic heap-sizing collectors. We chose the real memory allocations to reveal the effect of using large heaps in small allocations as well as small heaps in large allocations. In particular, we sought to evaluate the ability of our algorithm to grow *and* to shrink the heap, and to compare its performance to the static heap collectors in both cases.

We compare the performance of the collectors by measuring their estimated running time, derived from the number of instructions simulated. As mentioned in Section 3, we attribute 2,000 instructions to each minor page fault and 5 million instructions to each major page fault. For our adaptive semi-space collector, we use the threshold $t = 5\%$ for computing the footprint. For our adaptive Appel collector we use $t = 10\%$. (Appel completes in rather less time overall and since there are a number of essentially unavoidable page faults at the end of a run, 5% was unrealistic for Appel.)

5.1 Adaptive vs. Static Semi-space

Figure 8 shows the estimated running time of each benchmark for varying initial heap sizes under the SS collector. We see that for nearly every combination of benchmark and initial heap size, our adaptive collector changes to a heap size that performs at least as well as the static collector. The left-most side of each curve shows initial heap sizes and corresponding footprints that do not consume the entire allocation. The static collector under-utilizes the available memory and performs frequent collections, hurting performance. Our adaptive collector grows the heap size to reduce the number of collections without incurring page swapping. At the smallest initial heap sizes, this adjustment reduces the running time by as much as 70%.

At slightly larger initial heap sizes, the static collector performs fewer collections as it better utilizes the available memory. On each plot, we see that there is an initial heap size that is ideal for the given benchmark and allocation. Here, the static collector performs well, while our adaptive collector often matches the static collector, but sometimes increases the running time a bit. Only pseudojbb and _209_db experience this maladaptivity. We believe that fine tuning our adaptive algorithm will likely eliminate these few cases.

When the initial heap size becomes slightly larger than the ideal, the static collector's performance worsens dramatically. This initial heap size yields a footprint that is slightly too large for the allocation. The resultant page swapping for the static allocator has a huge impact, slowing execution under the static allocator 5 to 10 fold compared to modestly smaller initial heap sizes. Meanwhile, the adaptive collector shrinks the heap size so that the allocation completely captures the footprint and little page swapping occurs. By performing slightly more frequent collections, the adaptive collector consumes a modest amount of CPU time to avoid a significant amount of disk access time, thus reducing the running time by as much as 90%.

When the initial heap size grows even larger, the performance of the adaptive collector remains constant. However, the running time with the static collector decreases gradually. Since the heap size is larger, it performs fewer collections, and it is those collections and their poor reference locality that cause the excessive page swapping. Curiously, if a static collector is going to use a heap size that causes page swapping, it is better off using an excessively large heap size!

Observe that for these larger initial heap sizes, even the adaptive allocator cannot match the performance achieved with the ideal heap size. This is because the adaptive collector's initial heap sizing mechanism cannot make a perfect prediction, and the collector does not adjust to a better heap size until after the first full collection.

A detailed breakdown. Table 1 provides a breakdown of the running time shown in one of the graphs from Figure 8. Specifically, it provides the results for the adaptive and static semi-space collectors for varying initial heap sizes with _213_javac. It indicates, from left to right: the number of instructions executed (billions); the number of minor and major faults; the number of collections; the percentage of time spent handling minor faults; the number of major faults that occur within the first two collections with the adaptive collector; the number of collections before the adaptive collector learns ("warms-up") sufficiently to find its final heap size; and the running time with the adaptive collector as a percentage of the running time with the static collector.

We see that at small initial heap sizes, the adaptive collector adjusts the heap size to reduce the number of collections, and thus the number of instructions executed, without incurring page swapping. At large initial heap sizes, the adaptive mechanism dramatically reduces the major page faults. Our algorithm found its target heap size within two collections, and nearly all of the page swapping occurred during that "warm-up" time. Finally, it controlled the minor fault cost well, approaching but never exceeding 1%.

5.2 Adaptive vs. Static Appel

Figure 9 shows plots of the running time for each of our benchmarks using both the original, static, Appel collector and our modified, adaptive, Appel collector, over varying initial heap sizes and fixed allocations. The results are qualitatively similar to those for the adaptive and static semi-space collectors. For all of the benchmarks, the adaptive collector yields significantly improved performance for large initial heap sizes that cause heavy page swapping with the static collector. It reduces running time by as much as 90%.

For approximately half of the benchmarks, the adaptive collector improves performance almost as dramatically for small initial heap sizes. However, for the other benchmarks, there is little or no improvement. The Appel algorithm uses frequent nursery collections, and less frequent full heap collections. For our shorter-lived benchmarks, the Appel collector incurs only 1 or 2 full heap collections. Therefore, by the time that the adaptive collector "warms-up" to select a better heap size, the execution ends.

Notice also that, for the static collector, there are sometimes two local minima—heap sizes that provide improved performance when compared to adjacent heap sizes. The larger of these two heap sizes occurs when the nursery collections remove enough dead objects to prevent any full heap collections. This situation occurs for benchmarks with higher live sizes, such as _213_javac, _228_jack, and pseudojbb; it does not obtain for benchmarks with lower live sizes, such as _202_jess and _205_raytrace. Since a nursery collection visits much less of the heap, it does not exhibit the poor locality of a full heap collection, and thus does not cause large footprints that lead to page swapping.

Furthermore, our algorithm is more likely to be maladaptive when its only information is taken from nursery collections. Consider _228_jack at an initial heap size of 36MB. That heap size is sufficiently small that the static collector incurs no full heap collections. For the adaptive collector, the first several nursery collections create a footprint that is larger than the allocation, so the collector reduces the heap size. This heap size is small enough to force the collector to perform a full heap collection that references far more data than the nursery collections did. Therefore, the footprint suddenly grows far beyond the allocation and incurs heavy page swapping. The nursery collection leads the adaptive mechanism to predict an unrealistically small footprint for the select heap size.

Although the adaptive collector then chooses a much better heap size following the full heap collection, execution terminates before the system can realize any benefit. In general, processes with particularly short running times may incur the costs of having the adaptive mechanism find a good heap size, but not reap the benefits that follow. Unfortunately, most of these benchmarks have short running times that trigger only 1 or 2 full heap collections with pseudo-adaptive builds.

Parameter sensitivity. It is important, when adapting the heap size of an Appel collector, to filter out the misleading information produced during small nursery collections. Furthermore, because a maladaptive choice to grow the heap too aggressively may yield a large footprint and thus heavy page swapping, it is important to grow the heap conservatively. The algorithm described in Section 4.3 employs two parameters: the *conservative factor*, which controls how conservatively we grow the heap in response to changes in footprint or allocation, and the *nursery filter factor*, which controls which nursery collections to ignore.

We carried out a sensitivity test on these parameters. We tested all combinations of conservative factor values of {0.66, 0.50, 0.40} and nursery filter factor values of {0.25, 0.5, 0.75}. Figure 7 shows _213_javac under the adaptive Appel collector for all nine combinations of these parameter values. Many of the data points in this plot overlap. Specifically, varying the conservative factor has no effect on the results. For the nursery filter factor, values of 0.25 and 0.5 yield identical results, while 0.75 produces slightly improved running times at middling to large initial heap sizes. The effect of these parameters is dominated by the performance improvement that the adaptivity provides over the static collector.

Dynamically changing allocations. The results presented so far show the performance of each collector for an unchanging allocation of real memory. Although the adaptive mechanism finds a good, final heap size within two full heap collections, it is important that the adaptive mechanism also quickly adjust to dynamic changes in allocation that occur mid-execution.

Figure 10 shows the result of running _213_javac with the static and adaptive Appel collectors using varying initial heap sizes. Each plot shows results both from a static 60MB allocation and a dynamically changing allocation that begins at 60MB. The left-hand plot shows the results of increasing that allocation to 75MB after 2 billion instructions (2 sec), and the right-hand plot shows the results of shrinking to 45MB after the same length of time.

When the allocation grows, the static collector benefits from the reduced page faulting that occurs at sufficient large initial heap sizes. However, the adaptive collector matches or improves on that performance. Furthermore, the adaptive collector is able to increase its heap size in response to the increased allocation, and thus reduce the garbage collection overhead suffered when the allocation does not increase.

The qualitative results for a shrinking allocation are similar. The static collector's performance suffers due to the page swapping caused by the reduced allocation. The adaptive collector's performance suffers much less from the reduced allocation. When the allocation shrinks, the adaptive collector will experience page faulting during the next collection, after which it selects a new, smaller heap size at which it will collect more often.

Notice that when the allocation changes dynamically, the adaptive allocator dominates the static collector—there is no initial heap size at which the static collector matches the performance of the adaptive allocator. Under changing allocations, adaptivity is necessary to avoid excessive collection or page swapping during some phases of execution.

We also observe that there are no results for the adaptive collector for initial heap sizes smaller than 50MB. When the allocation shrinks to 45MB, page swapping always occurs. The adaptive mechanism responds by shrinking its heap. Unfortunately, it selects a heap size that is smaller than the minimum required to execute the process, and the process ends up aborting. This problem results from the failure of our linear model, described in Section 3.2, to correlate heap sizes and footprints reliably at such small heap sizes.

We believe we can readily address this problem in future work (possibly in the final version of this paper). Since our collectors can already change heap size, and since it is simpler for a collector to expand its heap than to contract it, we believe that a simple mechanism can grow the heap rather than allowing the process to abort. Such a mechanism will make our collectors even more robust than static

collectors that *must* abort if the heap size is too small.

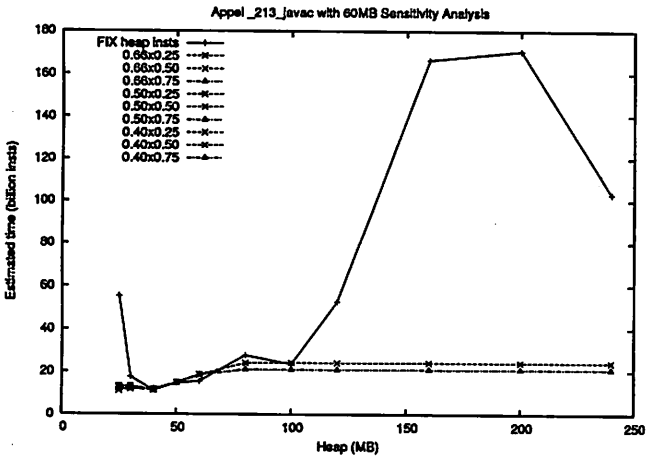


Figure 7: _213_javac under the Appel collectors given a 60MB initial heap size. We tested the adaptive collector with 9 different combinations of parameter settings, where the first number of each combination is the *conservative factor* and the second number is the *nursery filter factor*. The adaptive collector is not sensitive to the conservative factor, and is minimally sensitive to the nursery filter factor.

6. FUTURE WORK

Our adaptive collectors demonstrate the substantial performance benefits possible with dynamic heap resizing. However, this work only begins exploration in this direction. We are bringing our adaptive mechanism to other garbage collection algorithms such as mark-sweep. We seek to improve the algorithm to avoid the few cases in which it is maladaptive. Finally, we are modifying the Linux kernel to provide the VMM support described in Section 4.2 so that we may test the adaptive collectors on a real system.

Other research is exploring a more fine-grained approach to controlling the page swapping behavior of garbage collectors. Specifically, the collector assists the VMM with page replacement decisions, and the collector explicitly avoids performing collection on pages that have been evicted to disk. We consider this approach to be orthogonal and complementary to adaptive heap sizing. We are exploring the synthesis of these two approaches to controlling GC page swapping.

Finally, we are developing new strategies for the VMM to select allocations for each process. A process that uses adaptive heap sizing presents the VMM with greater flexibility in trading CPU cycles for space consumption. By developing a model of the CPU time required for garbage collection at each possible allocation (and thus heap size), the VMM can choose allocations intelligently for processes that can flexibly change their footprint in response. When main memory is in great demand, most workloads suffer from such heavy page swapping that the system becomes useless. We believe that garbage collected processes whose heap sizes can adapt will allow the system to handle heavy memory pressure more gracefully.

7. CONCLUSION

Garbage collectors are sensitive to heap size and main memory allocation. Too small a heap size will incur frequent collections while under-utilizing the available memory. Too large a heap size will cause the process to suffer from heavy page swapping as full heap collections rapidly reference nearly all of the process's pages. Somewhere between these extremes is an ideal heap size for a given allocation that collects just often enough to avoid page swapping.

Users cannot *a priori* select a heap size that is near that ideal. Furthermore, main memory allocations are not constant—they change dynamically as the multiprogrammed workload places varying demands on the VMM. Therefore, a collector must change heap size in response to changing allocations.

We present a dynamic adaptive heap sizing algorithm. We apply it to two different collectors, semi-space and Appel, requiring only minimal changes to the underlying collection algorithm to support heap size adjustments. For static allocations, our adaptive collectors match or improve upon the performance provided by the standard, static collectors in the vast majority of cases. The reductions in running time are often tens of percent, and as much as 90%. For initial heap sizes that are too large, we drastically reduce page swapping, and for initial heap sizes that are too small, we avoid excessive garbage collection.

In the presence of dynamically changing allocations, our adaptive collectors strictly dominate the static collectors. Since no one heap size will provide ideal performance when allocations change, adaptivity is necessary, and our adaptive algorithm finds good heap sizes within 1 or 2 full heap collections.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grant number CCR-0085792. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. We are also grateful to IBM Research for making the Jikes RVM system available under open source terms, and likewise to all those who developed SimpleScalar and Dynamic SimpleScalar and made them similarly available.

9. REFERENCES

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

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalepeño virtual machine. *IBM Systems Journal*, 39(1), Feb. 2000.

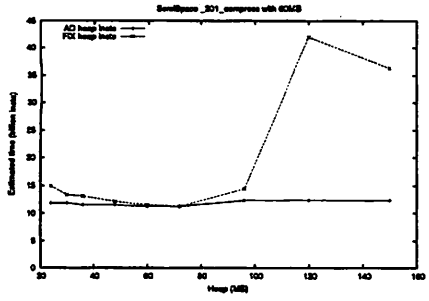
[3] B. Alpern, C. R. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, T. Ngo, M. Mergen, J. C. Shepherd, and S. Smith. Implementing Jalepeño in Java. In *Proceedings of SIGPLAN 1999 Conference on Object-Oriented Programming, Languages, & Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 314–324, Denver, CO, Oct. 1999. ACM Press.

[4] A. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, Feb. 1989.

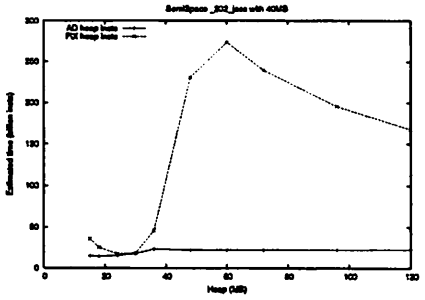
[5] O. Babaoglu and D. Ferrari. Two-level replacement decisions in paging stores. *IEEE Transactions on Computers*, C-32(12):1151–1159, Dec. 1983.

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

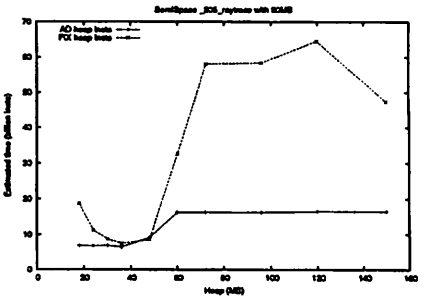
[7] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming,*



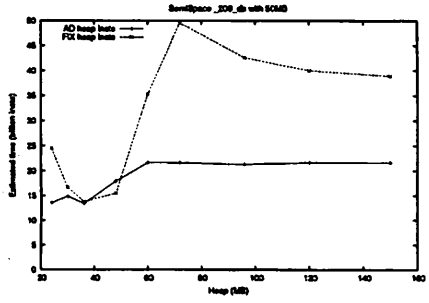
(a) _201_compress 60MB



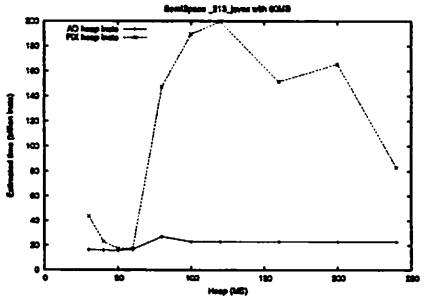
(b) _202_jess 40MB



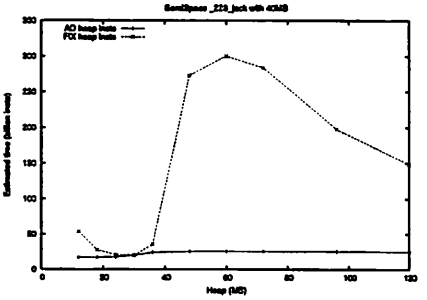
(c) _205_raytrace 50MB



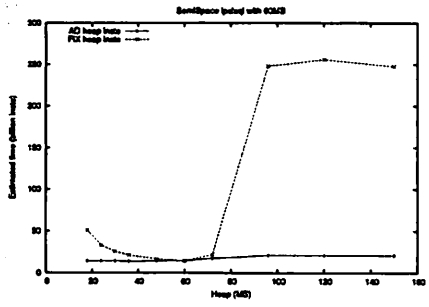
(d) _209_db 50MB



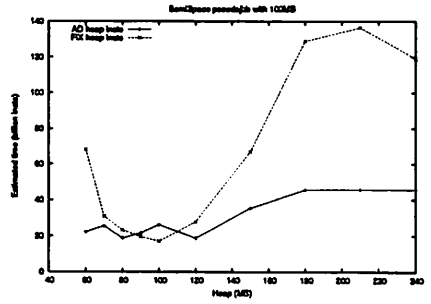
(e) _213_javac 60MB



(f) _228_jack 40MB



(g) ipsixql 60MB



(h) pseudojbb 100MB

Figure 8: The estimated running time for the static and adaptive SS collectors for all benchmarks over a range of initial heap sizes.

Systems, Languages & Applications, pages 353–366, Tampa, FL, June 2001.

[8] X. Huang, J. E. B. Moss, K. S. Mckinley, S. Blackburn, and D. Burger. Dynamic SimpleScalar: Simulating Java Virtual Machines. Technical Report TR-03-03, University of Texas at Austin, Feb. 2003.

[9] S. F. Kaplan, Y. Smaragdakis, and P. R. Wilson. Trace reduction for virtual memory simulations. In *Proceedings of the ACM SIGMETRICS 1999 International Conference on Measurement and Modeling of Computer Systems*, pages 47–58, 1999.

[10] S. F. Kaplan, Y. Smaragdakis, and P. R. Wilson. Flexible reference trace reduction for VM simulations. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 13(1):1–38, Jan. 2003.

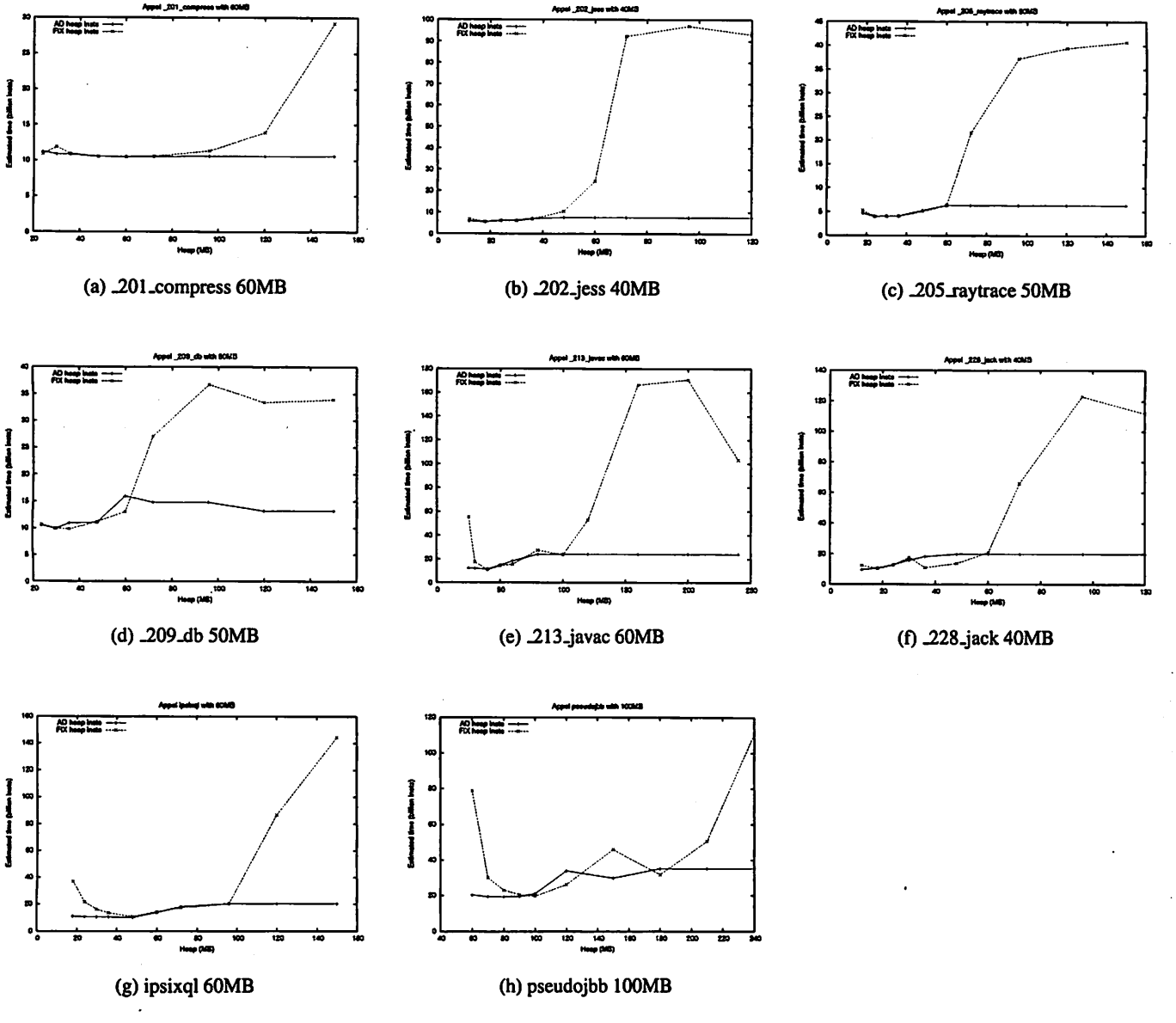
[11] K.-S. Kim and Y. Hsu. Memory system behavior of Java programs: Methodology and analysis. In *Proceedings of the ACM SIGMETRICS 2002 International Conference on Measurement and Modeling of Computer Systems*, volume 28(1), pages 264–274, Santa Clara, CA, June 2000.

[12] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson. The EELRU adaptive replacement algorithm. 53(2):93–123, July 2003.

[13] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of The 1999 USENIX Annual Technical Conference*, pages 101–116.

Heap (MB)	Inst's ($\times 10^9$)		Minor faults		Major faults		GCs		Minor fault cost		Hard faults 1st 2 GCs	Warm-up (GCs)	Ratio (AD/FIX)
	AD	FIX	AD	FIX	AD	FIX	AD	FIX	AD	FIX			
30	15.068	42.660	210,611	591,028	207	0	15	62	0.95%	0.95%	0	2	62.28%
40	15.251	22.554	212,058	306,989	106	0	15	28	0.95%	0.93%	0	1	30.04%
50	14.965	16.860	208,477	231,658	110	8	15	18	0.95%	0.94%	0	1	8.22%
60	14.716	13.811	198,337	191,458	350	689	14	13	0.92%	0.94%	11	1	4.49%
80	14.894	12.153	210,641	173,742	2,343	27,007	14	9	0.96%	0.97%	2236	1	81.80%
100	13.901	10.931	191,547	145,901	1,720	35,676	13	7	0.94%	0.90%	1612	2	88.92%
120	13.901	9.733	191,547	128,118	1,720	37,941	13	5	0.94%	0.89%	1612	2	88.63%
160	13.901	8.540	191,547	111,533	1,720	28,573	13	3	0.94%	0.88%	1612	2	85.02%
200	13.901	8.525	191,547	115,086	1,720	31,387	13	3	0.94%	0.91%	1612	2	86.29%
240	13.901	7.651	191,547	98,952	1,720	15,041	13	2	0.94%	0.87%	1612	2	72.64%

Table 1: A detailed breakdown of the events and timings for _213_javac under the static and adaptive SS collector over a range of initial heap sizes. *Warm-up* is the time, measured in the number of garbage collections, that the adaptivity mechanism required to select its final heap size.

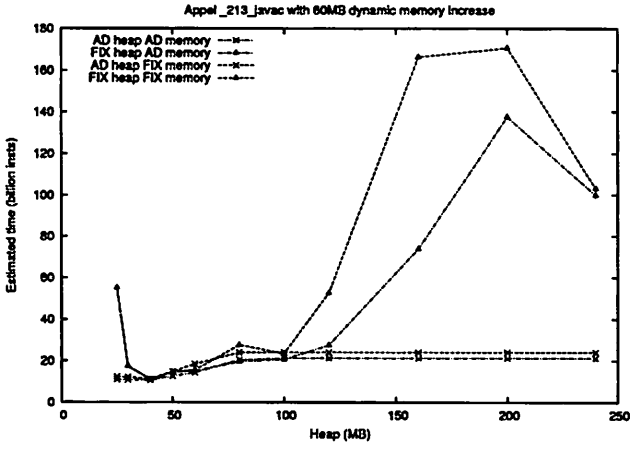


(a) _201_compress 60MB (b) _202_jess 40MB (c) _205_raytrace 50MB

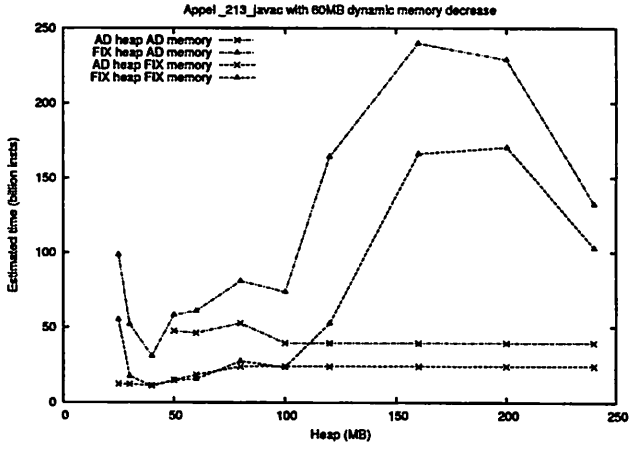
(d) _209_db 50MB (e) _213_javac 60MB (f) _228_jack 40MB

(g) ipsixql 60MB (h) pseudojbb 100MB

Figure 9: The estimated running time for the static and adaptive Appel collectors for all benchmarks over a range of initial heap sizes.



(a) _213_javac 60MB– >75MB



(b) _213_javac 60MB– >45MB

Figure 10: Results of running _213_javac under the adaptive Appel collector over a range of initial heap sizes and *dynamically varying* real memory allocations. During execution, we increase (left-hand plot) or decrease (right-hand plot) the allocation by 15MB after 2 billion instructions.