

# CRAMM: Cooperative Robust Automatic Memory Management

Emery D. Berger

J. Eliot B. Moss

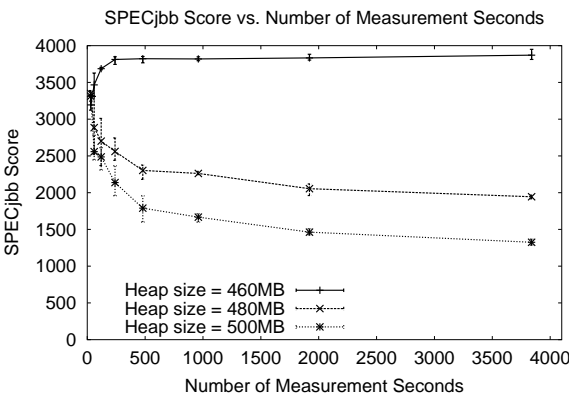
Scott F. Kaplan

## C Project Description

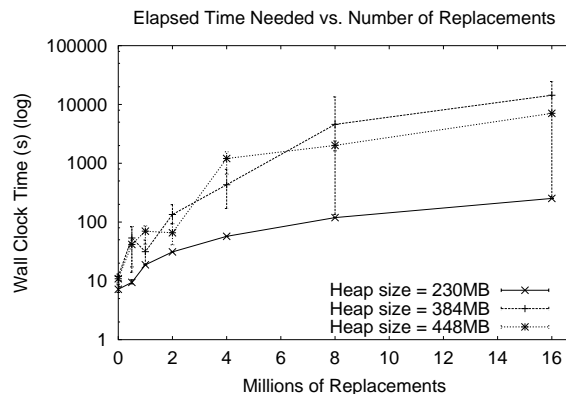
Programming languages that rely on garbage collection are becoming ubiquitous. Java and C# are especially popular. Both languages provide numerous software engineering advantages over languages like C and C++. The advantages of garbage collection include safety from accidental memory overwrites, protection from security violations, and the automatic prevention of space leaks. An important disadvantage of garbage collection is its negative impact on memory usage and execution time. Garbage-collected programs tend to consume more memory and exhibit worse locality than similar programs written in C and C++. They also suffer from reduced application throughput and increased response time due to garbage collection pauses.

The focus of almost all recent research on garbage collection has been on improving the performance of individual applications executing on a dedicated machine. This model is obsolete. On servers, there is increasing use of Java and C# for applications and servlets as well as for stored procedures in database management systems. On desktops, components of office suites like Microsoft Office are being written in C#, and StarOffice already includes Java-based extensions. In the near future, we expect that both servers and desktop machines will spend most of their time executing a multitude of garbage-collected applications.

We believe that this trend is leading us towards a performance disaster. Garbage-collected applications already require more memory than their counterparts written in C and C++, and interact poorly with virtual memory managers. A recent memo from Sun Microsystems cites the increased footprint of Java applications as one of the key problems with Java and “a barrier to the delivery of reliable



(a) An example of the effects of paging on SPECjbb, an enterprise workload running Java Beans. The “score” corresponds to the number of transactions processed (bigger is better). Throughput drops by nearly 2/3 as heap sizes exceed available memory.



(b) Execution time (smaller is better) for a synthetic benchmark with memory pressure simulated by increasing heap size. As heap size grows, execution time increases by orders of magnitude. Note that the execution time axis is shown in log scale.

Figure 1: Memory pressure on garbage-collected applications leads to significant performance degradation. For each graph, the error bars correspond to the maximum and minimum values over three runs.

applications” [Nic03]. The impending arrival of 64-bit architectures will double the size of pointers, further increasing application memory requirements. As typical footprints grow, fewer applications will fit in main memory, causing virtual memory paging to result. Because disk accesses are five to six orders of magnitude more expensive than RAM accesses, performance will plummet and response times will skyrocket.

The root of the problem is the *lack of cooperation between the garbage-collector and the virtual memory manager*. Modern garbage collectors rely on static heap sizes and policies that do not take memory pressure into account. Virtual memory managers do not communicate memory pressure to the garbage collector, and current garbage collectors lack any mechanism to respond. The virtual memory system can systematically choose to evict pages that the collector is about to touch. The collector’s activity disrupts the reference behavior tracked by the virtual memory manager and is likely to touch pages that are not resident in RAM. A full heap collection can therefore trigger massive paging and cause systems to become unresponsive for 30 seconds or longer. Surprisingly, prior research focused on reducing garbage-collection pause times ignores the enormous impact of paging.

Paging already has a significant impact on real application performance. Figure 1(a) depicts runs of SPECjbb [Cor00], a widely-used server-side benchmark of an enterprise workload running Java Beans. By increasing the garbage collector’s heap size, we simulate the decrease in available physical memory caused by competing applications. When the heap fits entirely in memory (a heap size of 460MB), the system is able to maintain a throughput of nearly 4000 transactions per second. When the heap size is increased beyond the limits of physical memory, throughput is reduced as the workload increases until it drops to around 1400 transactions per second. This dramatic drop-off in performance is surprising given that SPECjbb is not particularly allocation-intensive and requires few garbage collections.

For applications that do stress the garbage collector, the situation is far worse. We developed a synthetic benchmark in Java called *node-replace* that builds two large binary trees and replaces nodes in the second tree. Increasing the number of replacements increases the amount of data allocated, thus increasing the work required of the garbage collector. Because this benchmark makes intensive use of the garbage collector, these results directly reflect the impact of increasing memory pressure on garbage collection. Figure 1(b) illustrates the results of executing two instances of this application simultaneously on a system with 512MB of RAM. When both applications fit in memory (heap size of 230MB), the execution time is at most around 4 minutes. As memory pressure increases, execution time ascends to over 2 hours, a 30-fold increase.

We believe it is wishful thinking to expect to be able to simply buy our way out of this problem. On some servers, increasing physical memory requires adding additional processors, greatly increasing the expense of a memory upgrade. For instance, adding 4GB of RAM to a Sun Enterprise 10000 costs over \$70,000, or \$17,500 per gigabyte. Memory cost is also an increasingly large portion of the cost of desktop and laptop systems. Upgrading to 512MB can increase the cost of an entry-level PC by 20%, and going to 1GB increases it by 50%. These percentages rise to as high as 70% of the cost of laptop systems, and for some laptops, memory expansion is not even an option. Even when money is no object, the question remains: how much memory is enough? Just one unanticipated workload that exceeds available memory can bring a system to its knees.

## C.1 Overview of Approach

We propose a cooperative approach to attack this problem that we call **CRAMM: Cooperative Robust Automatic Memory Management**. CRAMM includes a new virtual memory manager (VM) and new garbage collectors (GCs) that will allow garbage-collected applications to perform well and when overloaded, to degrade gracefully. Figure 2 depicts a high-level view of CRAMM, indicating the in-

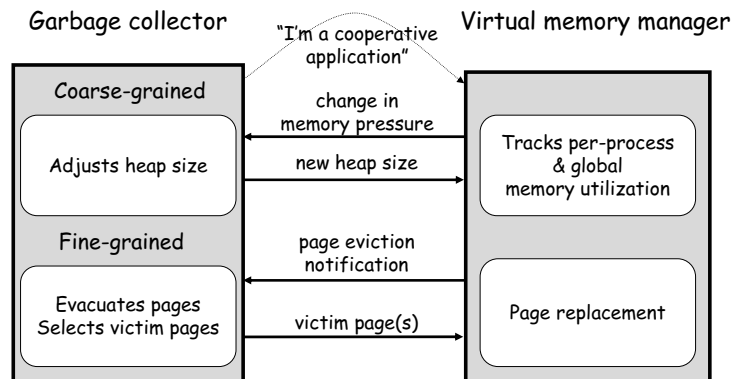


Figure 2: High-level view of CRAMM (Cooperative Robust Automatic Memory Management). On startup, the garbage collector registers itself with the operating system as a *cooperative* application. The application can then react to memory pressure messages by dynamically adjusting its heap size (**coarse-grained memory management**) and can cooperate to select victim pages for eviction (**fine-grained memory management**).

formation to be shared and the actions taken by each component. No existing virtual memory manager provides the functionality we require in order to permit this cooperation. We are developing a new virtual memory manager that can not only provide reliable, detailed information but also cooperate with garbage collectors to respond more effectively and flexibly to memory pressure.

On the garbage collection side, we are developing what we term **coarse-grained** and **fine-grained** memory management algorithms. Coarse-grained memory managers adjust application footprint dynamically in response to memory pressure. We will develop models of the interaction between application performance and heap size that will permit applications to evaluate the performance trade-offs of reducing or growing their heaps.

Fine-grained memory managers operate when paging is imminent. We have designed a preliminary fine-grained garbage collection algorithm that cooperates with the virtual memory manager to eliminate page faults that require disk accesses. Because coarse-grained and fine-grained memory management are complementary, we will develop and combine both approaches.

In addition to new virtual memory managers and garbage collectors, we will adapt and develop compiler analyses directed towards supporting and enhancing the virtual memory behavior of garbage-collected applications. Recent work has focused on compilation techniques directed towards improving cache performance. However, because paging is so costly, we can afford to perform more extensive analyses because their cost will almost always be outweighed by the savings obtained by avoiding paging. Among the analyses we will retarget towards this end are escape analysis, shape analysis, recurrence analysis, and region inference. We expect to exploit profile information when static analysis is insufficient.

We also intend to explore cooperative dynamic memory managers for non garbage-collected languages, including C and C++. However, we believe that the most significant performance results of this work will come from garbage-collected applications. We will leverage the additional flexibility provided by garbage-collected languages to improve application robustness under memory pressure, increase throughput, and reduce response time.

The remainder of this proposal details our approach. We first discuss our *GC-aware* virtual memory manager in Section C.2. We then discuss our two synergistic *VM-aware* garbage collector approaches. Section C.3 describes the coarse-grained garbage collector, which addresses footprint and virtual memory issues by changing the overall behavior of the application, including adjusting the heap

size. We include preliminary results that indicate directions that this research should take. We next address the fine-grained approach in Section C.4, which focuses on interaction for specific events. For example, when the virtual memory manager needs to reclaim a page from the application, we give the run-time system the opportunity to designate *which* page is taken. We discuss compiler approaches in Section C.5. We summarize our contributions in Section C.6, discuss the results of prior NSF support in Section C.7 and describe our project management plan in Section C.8.

## C.2 GC-Aware Virtual Memory Management

It is the responsibility of the VM to respond to *memory pressure*—a workload’s demand for memory resources. By having the VM cooperate with each GC in the workload, we believe it is possible for a system to respond more flexibly and effectively to memory pressure. Specifically, a GC is capable of changing the reference behavior of an application by changing its heap size. The dynamic resizing of the heap will require coordination between the VM and the GC. When the VM communicates memory utilization information to the GC, the collector can calculate a choice of heap size that is appropriate for its allocation. Specifically, the GC selects a heap size that will yield little or no page faulting behavior—one that would “just barely fit” in the allocation provided by the VM.

The selection of a heap size is critical for each GC. As depicted in Figure 3, a heap size that is too small will lead to more collections, thus reducing throughput and increasing *pause times*—interruptions in normal processing due to collection. A heap size that is too large may incur paging. The goal is to have each GC select *the largest heap size that causes no paging for its own allocation, incurs minimal collection overhead, and yet does not increase memory pressure on other processes*. Dynamically adapting to such a heap size online will require the GC to use the information provided by the VM to model the impact of different possible heap sizes, and then to select the best possible size.

### C.2.1 Prior Work

Unfortunately, existing virtual memory managers do not provide the support we require for this level of cooperation. These memory managers were developed 20 to 35 years ago [CH81, CO72, Den67, Den80, MGST70], when workloads were drastically different from those of today. Specifically, these policies were developed for batch processes that did not wait for user I/O, ran to termination, and exhibited a high degree of locality of reference due to the small main memory sizes available at the time. None of these characteristics is common for modern workloads, especially for garbage-collected applications.

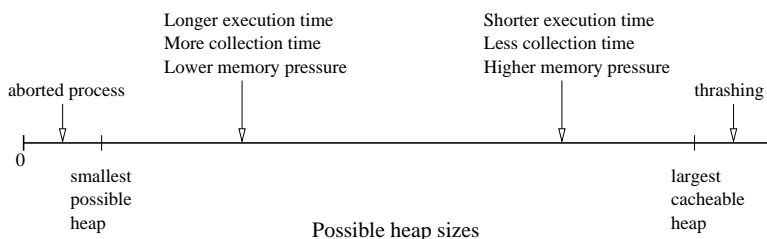


Figure 3: The range of possible heap sizes for a garbage collected application. The lower bound is a heap so small that the process must abort. The upper bound is a heap so large that thrashing results. Between those extremes, total execution time, pause times, and memory consumption vary. We seek to adjust heap sizes and allocations adaptively so that performance is as high as possible for each process given the memory pressure of the workload as a whole.

Current virtual memory managers fall into two categories: *global least recently used (gLRU)* [MGST70], used by most modern variants of Unix, and *Working Set* [Den67, Den80], a derivative of which is used in Windows. gLRU selects for eviction the page used longest ago among all pages from all processes, thus altering allocations for the processes that own the fetched and evicted pages. Unfortunately, gLRU provides no *isolation* between processes. A process that makes intensive use of its memory (e.g., during a full-heap garbage collection) can rob other processes of their allocations. Any process unlucky enough to lose its allocation is doomed to page, unable to run often enough to regain and maintain a reasonable allocation. On the other hand, Working Set provides *complete isolation* by keeping resident all those pages used in the last  $\tau$  references by each process, ignoring the activity of other processes. However, no single choice for  $\tau$  suffices for workloads with widely-varying locality characteristics such as the reference behavior of garbage-collected applications, which alternate between *mutator* (application) and collection phases.

An apparently-promising alternative policy is *Page Fault Frequency (PFF)* [MGST70]. PFF keeps the main memory miss rate of each process within a given range. If a process is faulting too much, its allocation is increased; if it faults too little, its allocation is decreased. PFF's greatest weakness is the high cost of its adaptive mechanism. To discover that a particular allocation is a poor choice for some process, PFF must first impose that allocation and then measure the result. This kind of blind adaptivity is unacceptable given the slowness of disks and the small amount of page faulting required to harm performance.

**Existing application/VM interfaces.** Some current Unix VM systems offer interfaces through which an application can communicate with the VM. For example, the `madvise()` system call allows an application to provide to the VM reference behavior hints (e.g., sequential traversal of a range of pages). There is no standard that dictates how a VM should respond to these hints. Another example is the `mincore()` system call that allows an application to determine whether a page is currently cached in main memory without incurring a page fault.

Although this interface may be useful to applications with rigorously regular reference patterns, it is insufficient for interaction with garbage collectors. The VM must be able to inform the GC of the current memory pressure, allocation decisions, and the application's reference behavior. The GC should be able to influence the VM's page replacement decisions, but not by forcing evictions in an untimely manner.

## C.2.2 A GC-Aware Virtual Memory Manager

Not only are the policies described above unable to deliver robust performance for modern workloads, they are not well suited for modification towards interaction with a VM-aware GC. As we will see in more detail in Section C.3, such a GC must not only be aware of its allocation, but also have sufficient information about its reference distribution so that it can calculate a new choice of heap size. At best, the existing policies could provide only *qualitative* hints to a GC, encouraging it to shrink or to grow. A VM should both adapt to modern workload characteristics and be able to provide a GC with sufficient *quantitative* information.

We describe here a new VM structure that will provide process isolation and will respond to memory pressure. It will maintain sufficient information about the reference behavior of each process so that the effects of new allocations can be projected, and so that GCs are able to make an informed selection of heap size.

In order to track the reference behavior of each process, the VM will maintain one LRU queue for each process. Additionally, there will be a *hit histogram* associated with each of these LRU queues

that records the number of references to pages at each queue position.<sup>1</sup> Each histogram quantifies the reference distribution of a process, indicating for any possible allocation the number of hits and misses that would have occurred. Since future reference behavior tends to resemble past reference behavior, these histograms will provide a good projection of upcoming memory demands. Previous adaptive virtual memory management mechanisms developed by one of the authors [KMC02, SKW99, WKS99] have relied on these histograms to predict future reference behavior.

Given the per-process reference distributions, the VM will perform a cost-benefit calculation, searching for a set of allocations to the processes that not only will maximize the system-wide hit rate, but will also achieve roughly equal hit rates across the processes if possible. Although searching for an optimal set of allocations is an intractable problem, we have developed fast approximations that yield nearly optimal results for reference distributions taken from real program reference traces. We will implement, evaluate, and refine these algorithms.

Implementation of these hit histograms requires care. Process footprints may be large, sometimes gigabytes each. However, histograms should consume little space, and should not contain so many entries that use of them requires substantial computation. Therefore, our VM implementation will group LRU queue positions and their pages into *bins*, where there will be one histogram entry per bin. By grouping anywhere from tens to hundreds of pages together, the grain of the histogram will be reduced, but so will its storage and computation requirements. We should also note that the histogram will not be maintained for some fraction of the *most* recently used pages. Because of the strong locality of reference, these pages are referenced too often for the VM to record their use. This lack of information is acceptable, as the heavy use of these pages implies that no reasonable VM would do anything but cache them.

Since applications tend to exhibit phase behavior, the VM will apply exponential decay to the histograms over time, thus ensuring that the information reflects recent reference behavior. Furthermore, a VM-aware GC can provide explicit markers for the beginning and ending of collection phases. At the start of a collection, the VM will preserve a copy of the hit histogram, since that histogram reflects the reference distribution of the mutator. During the collection phase, the VM can employ *compressed caching* [WKS99] to accommodate any increase in the set of active pages caused by the collector. The costs of page faults incurred by collection will be amortized by an aggressive use of *prepaging* [KMC02]. When the collection phase ends, the preserved hit histogram can be restored so that the VM has an accurate projection of the likely reference behavior of the reactivated application code. Note that this works in the face of the GC moving the objects onto different pages precisely because the histogram is in terms of LRU queue position, *not* in terms of virtual memory locations.

For those processes that include a VM-aware GC, the VM can then communicate the new allocation and reference distribution information for that process. As we discuss in Section C.3, the GC can use this information to perform its own cost-benefit analysis. Specifically, it will be able to derive an approximation of reference distributions that would occur with each different heap size, thus allowing it to select a new heap size consonant with its new allocation.

### C.3 Coarse-Grained VM-Aware Garbage Collection

One key aspect of *coarse-grained* garbage collection is that the GC can adjust an application's footprint dynamically in response to memory pressure. However, to do so sensibly, the GC requires a model of how footprint is affected by heap size and GC algorithm. Devising such a model, and using dynamic information from the VM at run time to drive it, is a challenge that is at the heart of the proposed

---

<sup>1</sup>That is, entry  $i$  in a histogram indicates the number of references to the  $i$ -th most recently used page.

research. Here we present some measurement results to illustrate the behaviors we will need to model, and outline some directions we will take to tackle the challenges in front of us.

To begin to grasp the relationships between application, GC algorithm, heap size, and footprint, we used a simulator to produce reference traces for some Java programs. The simulator is a user-mode functional simulator for the PowerPC called Dynamic SimpleScalar (DSS) [HBM<sup>+</sup>03]. DSS is based on SimpleScalar [BAB96] but is extended to handle additional features such as dynamic code generation, virtual memory mapping, and signals. We instrumented DSS to call an access tracing subroutine for each memory access. In order to decrease both simulation time and the size of the resultant traces, we reduce the traces using the Safely-Allowed Drop (SAD) algorithm developed by one of the authors [KSW99]. SAD yields a *reduced trace*, whose LRU page fetches and evictions are equivalent for real memories above a chosen value (128 pages = 512K bytes for our traces). We analyzed the reduced traces to develop curves showing the number of page faults for an LRU-based virtual memory manager (vertical) versus real memory size (horizontal). We ran the programs in IBM's Jikes RVM<sup>2</sup> [BCF<sup>+</sup>99], using several GC algorithms, each with a variety of heap sizes.

We note that running an application with different heap sizes results in different amounts of application work, so comparing solely on the basis of page faults would be misleading. We therefore started with the application running times for a given GC and heap size, measured on an otherwise idle machine with enough real memory to eliminate paging. To that application running time we added an estimated paging cost derived from our LRU model. Thus, our estimated time curves consist of a measured base application time for the specific program, GC, and heap size, plus paging time charged at 5 ms per fault.

Figure 4 shows sample results. The top two graphs show estimated running time for the SPECjvm benchmark program *jess* run with a mark-sweep (MS) collector (on the left), and with a generational copying (Gen) collector (on the right). The bottom two graphs show a breakdown of the MS curves into time spent in the mutator, i.e., running application code (on the left) and in the GC (on the right). Each graph has several curves, each corresponding to a different heap size. Note that the scales on all the graphs are the same, and that the vertical axis is logarithmic.

In interpreting these graphs, it may help to recall the general behavior of the GC algorithms. MS allocates space to objects until the heap is exhausted. It then recursively *marks* those objects transitively reachable from global (Java static) variables and thread stacks. After that, it sweeps through the heap, putting *unmarked* (collected) objects onto free lists. The collected space then becomes available for future allocations. A key property here is that the MS allocator always proceeds to consume all of the available heap space. This is evident in the graphs, in the steep drop for each curve near its right-hand end. Note that these drops occur at approximately the heap size (the value is a little different because there are memory regions accessed that are not part of the heap). Because the drop is so steep, it is clear that, for this collector, if the available memory is less than the knee point, we should either give the application more memory or have it reduce its heap size so as to move the knee to the left.

A second thing to notice is the asymptotic value reached by each curve (to the right): this value is lower for larger heap sizes, because the application does less collection work at larger heap sizes. If we plotted this asymptote for a range of heap sizes, it would have roughly a  $1/x$  shape, with the horizontal asymptote being the inherent application running time (given “infinite” memory) and the vertical asymptote being roughly the smallest heap size within which the application will run. The point is that there is a  $1/x$  shaped trade-off between total running time (in the absence of paging) and heap size.

---

<sup>2</sup>This research virtual machine for Java was formerly known as Jalapeño.

A third and surprising point is that at small *memory* sizes, larger *heap* sizes actually perform better. This performance improvement is primarily because they do less GC; i.e., in this region GC behavior dominates, as can be seen by inspecting the separated GC and mutator time graphs. Note that in this region, the process is paging heavily. However, under these conditions, using a larger heap size actually improves performance.

As a segue into discussing Gen, we note that the behavior of MS is quite clear and relatively predictable. We obtained similar graphs for other applications including *javac*. The locations and heights of the knees and crossover points vary depending on the application, but the general shape of the MS curves holds across applications. The Gen collector<sup>3</sup> divides the heap into two *generations*, called *young* and *old*. It allocates new objects into the young generation. When the young generation is full, it copies reachable young objects into the old generation, thus freeing up the young generation's space. Since the heap space is split between the two generations, and the old generation grows, the young generation gradually shrinks. When the young generations shrinks below a threshold value, we do a copying collection of all reachable objects in the old generation, creating a new, smaller, old generation. Again, a key property is that the allocator repeatedly allocates all available free space, then

<sup>3</sup>The particular algorithm we use is based on that of Appel [App87].

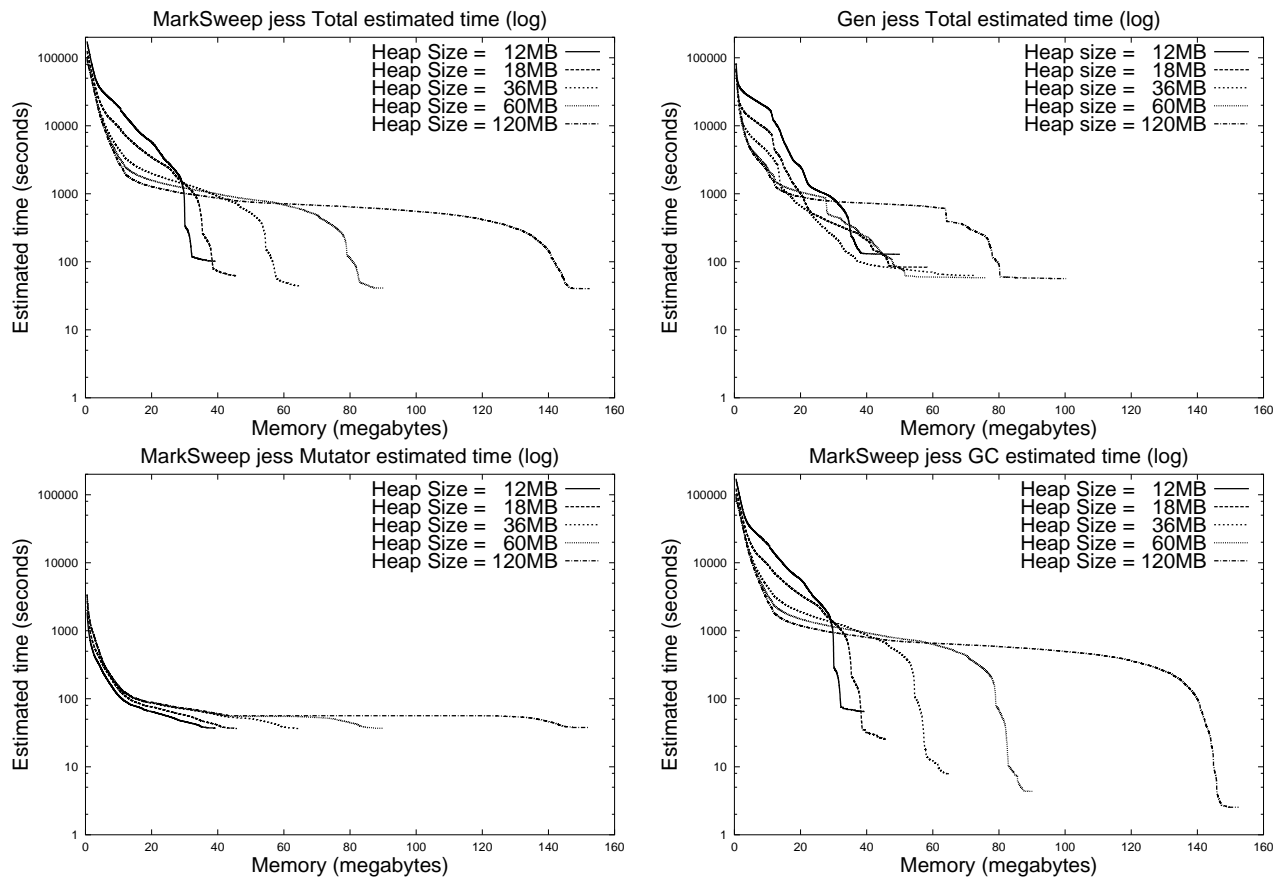


Figure 4: Estimated running time for the *jess* benchmark from SPECjvm98. The top graphs show total estimated time using mark-sweep and generational garbage collectors, while the bottom graphs separate out application and garbage collector activity using the mark-sweep collector. Notice that the impact of garbage collection determines the overall shape of the curve.

does some copying of reachable objects, compacting them into contiguous space. To ensure that there is room in which to do the copying, Gen allows only half the specified heap size to be allocated.

Considering the Gen graph, we see that there are similarities between Gen's behavior and what we see for MS, but the Gen curves are also clearly more complex. Some of this complexity has to do with the number of full-heap GCs Gen performs at each heap size (none at the largest heap size shown). We do get the same roughly  $1/x$  behavior of the asymptotes. We again see the same behavior that when the application is not given memory corresponding to its heap size, it pages heavily. Likewise, if we are in the region near the left end of the curves, we may want to *increase* heap size, so as to reduce GC frequency and thus reduce thrashing caused by GC. One difference between MS and Gen is that Gen degrades more gracefully: the rightmost drop in the curve is not always as steep as in MS.

**Previous coarse-grained studies.** Alonso and Appel [AA90] present one approach to VM/GC interaction: the GC queries the VM as to the amount of free space currently available, and uses that to adjust heap size. This approach is greatly limited by the *free memory* accounting within most VM systems: it is not a reliable indicator of the memory pressure of the current workload. For example, some pages may be little used, but not considered free, and will not be paged out until memory pressure forces their eviction. We believe we will obtain greater gains by more substantially changing the VM/GC interface. Alonso and Appel focused on shrinking heaps when memory pressure is high, but they ignored *expanding* heaps when memory pressure is low. Such expansion is important to reducing GC overhead when memory *is* available.

Brecht et al. [BALP01] adapt Alonso and Appel's approach to control heap growth, but in the absence of any VM signal they simply devise *ad hoc* rules for two given memory sizes. These memory sizes cannot change (i.e., this technique works only if the application is the only program in the system and the user provides the right memory size). Also, they used the Boehm-Weiser mark-sweep collector [Boe93], which can grow its heap but cannot shrink it. There are a few good ideas in the work, but it is not adaptive or cooperative in any way.

Kim and Hsu [KH00] use the SPECjvm98 benchmarks in examining the paging behavior of GC. Their strategy was to execute each program with a variety of heap sizes on a system with 32Mb of RAM. Their results are consistent with ours: 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. They conclude that there is an optimal heap size for each program for a given real memory. We agree, but argue that the real memory available to any given program changes dynamically, so we need to adjust the heap size in response.

**Coarse-grained VM-Aware GC.** Coarse-grained strategies for achieving VM-aware GC fall into two categories: adding intelligent adaptive heap size adjustment to *existing* GC algorithms, and inventing *new* GC algorithms that work even better with a GC-aware VM. To cooperate with a VM-aware GC algorithm, we need three things: to understand how the paging behavior changes with heap size and available memory, to predict the impact of heap size changes, and to carry out desired changes. We are already engaged in trying to understand, qualitatively and quantitatively, how a garbage-collected application behaves as we vary memory and heap size. For example, we have seen for some applications and for certain GC algorithms (MS among them), the VM must allocate to a process somewhat more memory than its heap size to contain the last knee of the paging curve. Above we mentioned several other properties evident from our preliminary results. We need to expand these results, considering more applications, more collectors, and a wider range of heap sizes to see whether the patterns we have seen so far are borne out and what additional patterns emerge. Of course, we need to study any new GC algorithms in the same way.

Predicting behavior, and in fact just knowing in what behavioral region an application is currently

executing given run-time measurements, is another significant research problem. We have some preliminary ideas on attacking this problem. Given separate measurements for mutator and GC phases of program execution, we can readily determine a few important things. One is whether GC is consuming a large fraction of the application's time. If not, we can likely shrink the heap some without dramatically increasing CPU time. Conversely, if GC is consuming a large fraction, *and* we are paging heavily, then we may be in the leftmost region of the curves, and we should *increase* our heap size.

Clearly, we will be interested in whether the program is being slowed much by paging, and if so, whether the paging is worse in the mutator or the GC phase. For generational collectors, we will distinguish minor and major collections as different phases. We will also estimate the size of a program's live data, a fact we know accurately after a full GC, and only approximately otherwise, but which is important in judging the smallest feasible heap size. On the one hand, it is challenging to estimate the entire paging curve for an application running under a particular GC, especially given that its access patterns and volume of live data are constantly in flux. On the other hand, not only do we believe that we can estimate some parameters reasonably well, and use them to adjust heap and memory size effectively, but we further believe that the LRU hit histograms provide useful run-time measurements that contain in a different form the same information as the curves we have plotted here. Thus we can develop analyses in the GC that examine the shape of the histogram to determine the current behavioral region and how best to adjust heap size for the future.

Changing the heap size dynamically, in particular *shrinking* the heap, is not easy with every existing GC algorithm. In some algorithms and circumstances, one might need to do a full compaction or copying of a heap to substantially reduce the memory in use. In such a case, the GC may need to negotiate with the VM for a temporary increase in space adequate to accomplish a full GC relatively quickly, and then drop its heap size once it has compacted. Again, this requires us to estimate how long such a GC will take and how much memory it will need. Note that this may be a good time to use compressed pages, since the excess demand is known to be temporary.

What about new GC algorithms? Much GC work of the past two decades has focused on achieving improved throughput, with little consideration of either footprint or impact in a multiprogrammed environment. In the proposed work we will design new GCs towards new goals: *predictability* of footprint given heap size, *adjustability* of heap size, and memory *utilization*, in addition to CPU overhead. For example, by these measures MS and non-generational copying collection are better than Gen in terms of predictability, and copying collection is better than MS in terms of adjustability. In general, a judgment of adjustability should also include how *smoothly and cheaply* an algorithm can adjust heap size. Copying collection does not fare as well as MS on memory utilization, by which we mean how *well* it uses the memory given to it, not simply whether it uses it. We also need a quantitative sense of the marginal impacts of adjustments in memory or heap size, e.g., if we shrink the heap by so much, how much larger will the GC time be? Clearly there are tradeoffs between the various goals, including CPU overhead. In a multiprogrammed environment we may want an adjustable CPU overhead parameter for GC, which would be set to a low value when memory is plentiful, but a higher value as memory gets tight, so that the GC will see that a smaller heap, with consequently more frequent GC, is acceptable. Ultimately this involves cooperative management of both CPU and memory resources between the VM, scheduler, and the applications.

#### **C.4 Fine-Grained VM-Aware Garbage Collection**

By working closely with the virtual memory manager, a *fine-grained* garbage collector can eliminate page faults by taking advantage of detailed information about the layout of data and the reference patterns of its collection algorithms. This approach complements coarse-grained garbage collection,

reducing or eliminating page faults when the heap cannot fit in the allocated space and when shrinking the heap would induce many garbage collections. Both coarse- and fine-grained techniques will be necessary for a memory system that can robustly support a wide range of workload characteristics.

#### C.4.1 Previous Fine-Grained Studies

It has been thirty years since Fenichel and Yochelson modified garbage collection to work in VM systems [FY69], and there has followed much research into GC algorithms [WJNB95, JL96]. However, after two early attempts to improve the paging behavior of GC in VM environments [Bae72, Moo84], there have been few studies of cooperative approaches. There have been several indirect attempts to improve GC paging performance by improving the garbage collector's locality [Cou88, WLM91b, CL98, Ste99, CDL99, CLH99, TO01]. Although these studies demonstrate modest improvements in the total number of page faults, they present results generated on isolated machines and do not consider the effects of varying memory pressure for multi-programmed workloads.

Cooper, Nettles, and Subramanian [CNS92] exploited *external pagers* in the Mach operating system [MA90] to allow the GC to influence VM decisions, but their approach contained significant limitations. First, the external pagers provided an ill-fitting interface, requiring the GC to become an *ad hoc* main memory manager. Specifically, the GC and external pager aggressively flushed *discardable* pages—ones containing no live data—at times when it may not have been necessary, thus degrading performance. Also, the external pager had to act as an intermediary cache that stored pages evicted by the VM but identified by the GC as *non-discardable*, as they still contained live data. Not only is this interface problematic, but merely determining whether a page is discardable does not take full advantage of the GC's capability to manage its space. The second limitation was that the study did not examine multi-programmed workloads, and thus does not reflect the need for the VM and GC to respond to changes in memory pressure.

#### C.4.2 Fine-grained VM-Aware GC

Our fine-grained VM-aware GC algorithms will eliminate page faults by using their knowledge of the heap to guide the VM in its page replacement decisions. The GC-aware VM will communicate with the GC when a page must be selected for eviction. If there are no discardable pages available, the GC algorithm can find a nearly empty page and then evacuate its remaining data, thereby making it discardable. The discardable page can then be provided to the VM for eviction. The GC can further indicate to the VM that this page is discardable so that, upon any future reference to the page, the VM can provide a zero-filled page, thus avoiding a needless disk read operation.

We have designed an API for interactions between the GC and VM that is simple yet sufficiently powerful to provide the communication necessary to achieve our goals. The API is independent of any one GC algorithm so that we can implement it within any VM-aware GC. When a process containing a VM-aware GC begins, it must register with the VM, providing the addresses of upcall functions that allow the VM to initiate communications. When processing a page fault that requires a page replacement decision, the VM can call the GC to request a page selection for eviction. The GC will provide to the VM, in decreasing order of desirability:

- **A discardable page**, either already available or created by evacuation.
- **A non-discardable page** that the GC projects will not be used soon.
- **No page**, thus leaving the VM to make its own selection.

Although crossing the kernel/user boundary to perform this call will require some computation time, its effect on performance should be negligible. Under normal circumstances, few page faults will be occurring, and this upcall will be performed infrequently. If page faulting becomes frequent, the cost of disk accesses will far exceed the cost of upcalls.

We must note that it is not a simple matter for the GC to select a non-discardable page projected not to be used soon. We have developed an algorithm to solve this problem, but its effectiveness will need to be evaluated experimentally. We consider it a substantial research problem for a GC to predict the reference behavior of its collection phases.

The operating system will also provide system calls that a GC can use to initiate communication with the VM. Specifically, the GC may request from the VM a copy of the process's LRU queue and hit histogram. The GC will use this information not only to decide which page to evict, but also to schedule a collection or to decide upon which region to garbage collect. Similarly, prior to performing a collection, the GC can call the VM to inform it of the pages that it anticipates referencing soon as part of collection activity.

Again, one of the substantial research tasks will be the experimental evaluation and further development of our algorithms that will allow a GC to predict its own collection behavior, as well as to adapt that behavior to the current VM state. Note also that the VM will be modified to take advantage of the information provided by the GC, performing prepaging to make resident the pages indicated for collection, and using non-LRU page replacement selections to keep them resident.

## C.5 Compiler Involvement

There are a number of ways in which compilers, in their analyses, optimizations, and code generation techniques, can support and enhance VM-GC cooperation.<sup>4</sup> One way is to develop knowledge of the allocation and lifetime patterns exhibited by a program. For example, *escape analyses* [Bla98, Bla99] can identify some allocation call sites that allocate objects that do not outlive the method invocation during which the objects are allocated. These analyses can also identify objects that do not escape their allocating Java thread (i.e., remain private to that thread). Both analyses may allow us to improve the density of useful objects in virtual memory, as well as to reduce the footprint of many invocations of the GC. *Region inference* [TB98] is a similar analysis, which bounds the stack frame to which an allocated object may escape. Both escape analyses and region inference may be augmented by profiling to determine the actual volume of objects that escape to various levels. Profiling is complementary in that it may find that no objects from certain allocation sites escape, even though escape analysis could not *prove* it. We can use both analysis results and profile information to guide allocation activity.

*Shape analysis* [SRW98] is another potentially useful compiler analysis. It provides information about the organization of a data structure, which can support further analyses that extract access patterns (e.g., following a chain of pointers in a given “direction” on a linked data structure). An example of such analysis is Cahoon's *recurrence analysis* [CM01b]. Knowledge of the shape of a data structure and of patterns of access to the structure can drive *object clustering*, at allocation time and during collection. In particular, copying GCs can *recluster* objects. To date, much work in this area has focused on improving cache performance (for example, [CL98]), though there is some earlier work addressing virtual memory performance [WLM91a]. This work is more than due for reconsideration and updating, in the context of modern languages and run-time environments.

Another potential direction for compiler and profiling work is *pointer swizzling* [Mos92]. The point here is to maintain a cache of currently-used objects. The cache deals in terms of a smaller log-

---

<sup>4</sup>In this section, we supply representative citations rather than a comprehensive list of citations for each topic.

ical address space (e.g., 32 bits), while the full space of objects is larger (e.g., 64 bit address space). Swizzling allows references within actively used objects to be smaller, saving considerable space. Effective and efficient swizzling requires a suitable run-time system, and profiling and/or static compiler analysis to swizzle in batches and reduce overheads.

More speculatively, we can imagine maintaining “referenced” and “modified” bits at the level of objects, using compiler and run-time system support. This would allow dynamic clustering of objects based on reference patterns as they are observed at run time.

Many of the techniques we describe above have proved challenging to implement so as to extract significant benefits in *cache performance*. However, because paging is so costly, applying them to improve *paging performance* is much more promising. Put another way, we can afford more application CPU time overhead to avoid paging and achieve better utilization of real memory than we can afford to make modest improvements in cache performance.

## C.6 Summary

The popularity of garbage-collected programming languages and current architectural trends are on a collision course that will, if unchecked, produce a performance disaster for both desktop and server systems. We attack this problem of tremendous academic and societal importance by designing cooperative systems support for modern garbage-collected languages. We expect to produce not only knowledge and techniques but also prototype systems that exhibit these techniques to the community at large. Further, we will train a new generation of researchers to conceive and build these systems. We start with a problem of great relevance that incorporates many research challenges in areas where we have the expertise, and, if funded, the resources to make excellent progress. In summary, we are proposing a radical shift in the design of operating systems and programming language run-time systems, and we expect to produce systems that make languages such as Java and C# outperform programs written in non-garbage-collected programming languages.

## C.7 Results of Prior NSF Support

**J. Eliot B. Moss, Kathryn S. McKinley, and Charles C. Weems; NSF CCR-0085792; \$3,156,901; 9/1/2000–8/31/2005; “ITR/SW: Dynamic Cooperative Performance Optimization”.** This grant has focused substantially on memory performance, but from the standpoint of interactions between the architecture and the language implementation (cache performance, and to some extent, translation look-aside buffer performance), as opposed to interactions with the operating system or virtual memory management (paging). It is clear that paging can have a more dramatic impact than virtually *any* of the optimizations or algorithms being explored under this prior funding, yet substantial investigation of operating system interactions lies outside the scope of the previous award’s work. We further note that for PI Moss, the direct support of the previous award consists of summer salary, two research assistants focused on that work, and partial support for a staff programmer. There are no financial resources to undertake the work proposed here.

The goal of the (previous) project is to develop performance optimizations for Java that can be applied *dynamically* (while programs run) and that involve *cooperation* between system components, particularly between the compiler and the GC, or between the architecture and the collector. While we focus on time performance, we are also starting to consider issues of power consumption by Java virtual machines. We are particularly interested in improving cache and GC performance of Java programs, and envision feeding back cache performance information at run time, to drive adaptive optimizations that will improve performance as a program runs. We are a leading team in GC and memory management innovation and performance evaluation.

Project publications so far include a Ph.D. dissertation [Sin02], 4 journal articles [DMM01, HDH02, SMS02, HMar], and 21 conference/workshop papers [UWK<sup>+</sup>01, WMR01, HM01, BZM01, SMM00, BMBW00, HMSW00, HDH01, HHDH02, PLDM02, BDH02, SHB<sup>+</sup>02, HBM<sup>+</sup>02, BJMM02, BM02, BSH<sup>+</sup>01, HIM02, BZM02, WMRW02, CM01a, HWM01]. We believe that these publications, even by their titles and venues, reflect the scope, variety, energy, and quality of the ongoing work of this project, and so for the sake of space do not go into detail here. The project now extends to five sites. We manage it by having weekly bi-lateral and multi-lateral phone meetings for each ongoing line of work, and by gathering at least twice a year for two days of face to face meetings.

**Education and Human Resources Impact:** This award has had affiliated with it (so far): 8 faculty, 1 post-doctoral associate, 28 graduate students, 4 undergraduate students, and 2 technicians (programmers). Clearly it is having a significant impact in training people to undertake performance evaluation and optimization of systems built with modern programming languages, such as Java.

**Additional Results from Prior Research:** We are proposing substantial prototype work, so we mention some previous systems we have built, citing related publications, to show our system-building credentials for research:

- We built a complete **Smalltalk interpreter** from scratch [GR83, Mos87], used with several systems below.
- We built **Mneme**, a lightweight persistent object store, from scratch [MS88, Mos89, Mos90]. We used it to implement Persistent Smalltalk (below) and also as an experimental platform in its own right, producing results related to pointer swizzling [Mos92] and the efficiency of using an object store to support full-text information retrieval [BCCM94, BCC94], leading to a PhD dissertation [Bro95].
- By augmenting the Smalltalk interpreter, we built **Persistent Smalltalk** [HMB90, Hos91, MH94] and measured such things as the cost of object fault handling [HM93a], the cost of detecting and writing back changes to objects [HBM93], and the (non)desirability of using virtual memory protection traps to drive persistence mechanisms [HM93b], leading to a PhD dissertation [Hos95].
- We designed and built the **UMass Language-Independent Garbage Collector Toolkit** [HMDW91]. We have used it to explore heap behavior for Standard ML of New Jersey programs [Ste93], to explore the costs of various garbage collector write barriers [HMS92], and of a new hybrid write barrier scheme [HH93]. Related investigations of GC costs produced interesting insights concerning the interactions of certain memory management strategies with various cache hardware [DTM94, DTM95, TD96].
- We reimplemented the GC Toolkit from scratch for the Jikes RVM, and it is available under an open source license. It has been very useful in developing new GC algorithms in the Jikes RVM, and has helped spur a second generation Jikes RVM memory management toolkit called **JMTk**, now part of the standard Jikes RVM release.
- We extended and experimented with a **Modula-3 compiler**, exploring optimizations relevant to strongly typed object-oriented languages, leading to papers [DMM96, DMM98] and a PhD dissertation [Diw97].

In sum, we have a long track record of building significant prototypes and exploiting them in experimental research, reported in conferences and journals, and leading to PhD dissertations.

## **C.8 Project Management Plan**

The project involves collaboration between two institutions and sites, UMass, a PhD granting, land-grant university, and Amherst College, a distinguished institution devoted to undergraduate education. In many respects, managing the collaboration is simple, because we are all in the same town, only a mile or two apart. Thus we can have weekly group meetings together, face-to-face. We will house some equipment at Amherst College, for the convenience of the project staff there (Prof. Kaplan plus an undergraduate). We will invite the undergraduate staff to reside in the UMass lab for the summers, which will enhance their learning and interaction with the UMass graduate students, staff programmer, and faculty. We have good network connectivity between the institutions, so it is reasonable to manage code from UMass (e.g., using remotely accessible CVS repositories), and to run experiments on each others' machines.

Because the project encompasses several different kinds of work, each with somewhat complex pieces of software (Linux kernels, garbage collectors in Jikes RVM, architectural simulators, trace analyzers, and compiler optimizations), the issues of coordination are more internal than cross-site. The requested staff programmer time is important in accomplishing this coordination effectively, and in having the project's products be distributed and usable by others; see also the budget notes. We further observe that the number of research assistants requested is consistent with the variety of pieces of software with which we will work.

### **C.8.1 Plan of Work**

We now present a plan of work for accomplishing the research described above. In the first year, we will extend the Dynamic SimpleScalar system (DSS) to simulate a multiprogrammed CPU, including a CPU scheduler, a virtual memory simulator, and a paging model. Over the five year period, we will use DSS for building the models needed to perform coarse-grained garbage collection and for investigating garbage collector designs. We will first conduct extensive studies of existing garbage collectors and applications using this simulator. Also in Year 1, we will strip out the existing Linux virtual memory system and build our virtual memory manager. We will develop initial VM support for both coarse-grained and fine-grained collection. We will also implement non-concurrent versions of both garbage collection algorithms, which we expect to connect to the virtual memory manager by the end of the year. We will also develop and test a cooperative dynamic memory manager for C and C++ applications.

In Year 2, we will extend the virtual memory manager to provide further support for cooperative garbage collection. Using the insights obtained from our simulation studies, we will derive models of how footprint is affected by heap size and GC algorithms. We will incorporate these models into our coarse-grained garbage collector and perform experimentation using real workloads. Also in Years 2 and 3, we will develop concurrent versions of both of our GC algorithms to make these suitable for server applications. Year 3 will focus on empirical analysis of all aspects of the system as well as the development and incorporation of compiler analyses. Inspired by these results, we will investigate new algorithms that will further improve overall performance and robustness. Our work in Years 4 and 5 is hard to predict, but we envision new insights arising from our experiences with these algorithms on actual systems that will lead to refinement and extensions to our approach. The final result of our work will be an integrated memory management system that provides robust support for garbage collected applications, new virtual memory management algorithms, new garbage collection algorithms, new compiler analyses, and new OS features.

## D References

- [AA90] Rafael Alonso and Andrew 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.
- [App87] A. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25:275–279, June 1987.
- [BAB96] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: The SimpleScalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin-Madison, 1996.
- [Bae72] H. D. Baecker. Garbage collection for virtual memory computer systems. *Communications of the ACM*, 15(11):981–986, November 1972.
- [BALP01] Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang 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, Systems, Languages & Applications*, pages 353–366, Tampa, FL, June 2001.
- [BCC94] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, Santiago, Chile, September 1994. Morgan Kaufmann.
- [BCCM94] Eric W. Brown, James P. Callan, Bruce Croft, and J. Eliot B. Moss. Supporting full-text information retrieval with a persistent object store. In *Proceedings of the Fourth International Conference on Extending Database Technology (EDBT '94)*, pages 365–378, Cambridge, UK, March 1994.
- [BCF<sup>+</sup>99] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, United States, June 1999. ACM.
- [BDH02] Martin Burtscher, Amer Diwan, and Matthias Hauswirth. Compiler support for load-value prediction. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 222–233, Berlin, Germany, June 2002.
- [BJMM02] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: getting around garbage collection gridlock. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 153–164, Berlin, Germany, June 2002.
- [Bla98] Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 25–37, Montreal, June 1998. ACM Press.
- [Bla99] Bruno Blanchet. Escape analysis for object oriented languages: Application to Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 20–34, Denver, CO, October 1999. ACM Press.
- [BM02] Stephen M. Blackburn and Kathryn S. McKinley. In or out? Putting write barriers in their place. In *International Symposium on Memory Management*, pages 175–184, Berlin, Germany, June 2002. ACM.

- [BMBW00] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, Cambridge, MA, November 2000. ACM.
- [Boe93] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–206, 1993.
- [Bro95] Eric W. Brown. *Execution Performance Issues in Inference Network-Based Information Retrieval*. PhD thesis, University of Massachusetts, Amherst, 1995.
- [BSH<sup>+</sup>01] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinley, and J. Eliot B. Moss. Pretenuring for Java. In *Proceedings of the ACM 2001 Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 342–352, Tampa Bay, FL, 2001.
- [BZM01] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 114–124, Salt Lake City, UT, June 2001.
- [BZM02] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the ACM 2002 SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, Seattle, WA, November 2002.
- [CDL99] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming language design and implementation*, pages 13–24, May 1999.
- [CH81] Richard W. Carr and John L. Henessey. WSClock – a simple and effective algorithm for virtual memory management. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–95, December 1981.
- [CL98] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the First International Symposium on Memory Management*, pages 37–48, October 1998.
- [CLH99] Trishul M. Chilimbi, James R. Larus, and Mark D. Hill. Cache-conscious structure layout. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, May 1999.
- [CM01a] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, Barcelona, Spain, September 2001.
- [CM01b] Brendon Cahoon and Kathryn S. McKinley. Tolerating latency by prefetching Java objects. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Barcelona, Spain, September 2001.
- [CNS92] Eric Cooper, Scott Nettles, and Indira Subramanian. Improving the performance of SML garbage collection using application-specific virtual memory management. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming*, pages 43–52, San Francisco, CA, June 1992.

- [CO72] W. W. Chu and H. Opderbeck. The page fault frequency replacement algorithm. In *AFIPS Conference Proceedings*, volume 41(1), pages 597–609, Montvale, NJ, 1972. AFIPS Press.
- [Cor00] The Standard Performance Evaluation Corporation, 2000.
- [Cou88] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9), September 1988.
- [Den67] Peter J. Denning. The working set model for program behavior. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 15.1–15.12, January 1967.
- [Den80] Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.
- [Diw97] Amer S. Diwan. *Understanding and Improving the Performance of Modern Programming Languages*. PhD thesis, University of Massachusetts, Amherst, November 1997.
- [DMM96] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, (OOPSLA '96)*, pages 292–305, San Jose, CA, October 1996. ACM.
- [DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (OOPSLA '98)*, pages 106–117, Montreal, Quebec, June 1998. ACM.
- [DMM01] Amer S. Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Using types to analyze and optimize object-oriented programs. *ACM Transactions on Programming Languages and Systems*, 23(1):30–72, January 2001.
- [DTM94] Amer Diwan, David Tarditi, and J. Eliot B. Moss. Memory subsystem performance of programs using copying garbage collection. In *Conference Record of the Twenty-First ACM Symposium on Principles of Programming Languages*, pages 1–14, Portland, Oregon, January 1994.
- [DTM95] Amer Diwan, David Tarditi, and Eliot Moss. Memory system performance of programs with intensive heap allocation. *IEEE Transactions on Computer Systems*, 13(3):244–273, August 1995.
- [FY69] R. R. Fenichel and J. C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.
- [GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
- [HBM93] Antony L. Hosking, Eric Brown, and J. Eliot B. Moss. Update logging for persistent programming languages: A comparative performance evaluation. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pages 429–440, Dublin, Ireland, August 1993. Morgan Kaufmann.
- [HBM<sup>+</sup>02] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanović. Error-free garbage collection traces: How to cheat and not get caught. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2002)*, pages 140–151, Marina Del Ray, CA, June 2002.

- [HBM<sup>+</sup>03] Xianglong Huang, J.Eliot B.Moss, Kathryn S. Mckinley, Steve Blackburn, and Doug Burger. Dynamic SimpleScalar: Simulating Java virtual machines. Technical Report TR-03-03, University of Texas at Austin Department of Computer Science, February 2003.
- [HDH01] Martin Hirzel, Amer Diwan, and Antony Hosking. On the liveness accuracy of garbage collection. In *European Conference on Object-Oriented Programming*, June 2001.
- [HDH02] Martin Hirzel, Amer Diwan, and Johannes Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(6):593–624, 2002.
- [HH93] Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards, October 1993. Position paper for OOPSLA '93 Workshop on Memory Management and Garbage Collection.
- [HHDH02] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. Understanding the connectivity of heap objects. In *International Symposium on Memory Management*, pages 36–49, Berlin, Germany, June 2002. ACM.
- [HIM02] Matthew Hertz, Neil Immerman, and J. Eliot B. Moss. Framework for analyzing garbage collection. In *Foundations of Information Technology in the Era of Network and Mobile Computing: IFIP 17th World Computer Congress - TC1 Stream (TCS 2002)*, pages 230–241, Montreal, Canada, August 2002. Kluwer.
- [HM93a] Antony L. Hosking and J. Eliot B. Moss. Object fault handling for persistent programming languages: A performance evaluation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 288–303, Washington, DC, October 1993.
- [HM93b] Antony L. Hosking and J. Eliot B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 106–119, Asheville, North Carolina, December 1993. *ACM Operating Systems Review* 27, 5 (December 1993).
- [HM01] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. In *Proceedings of ACM 2001 Java Grande Conference*, pages 48–57, Palo Alto, CA, 2001.
- [HMB90] Antony L. Hosking, J. Eliot B. Moss, and Cynthia Bliss. Design of an object faulting persistent Smalltalk. COINS Technical Report 90-45, University of Massachusetts, Amherst, MA 01003, May 1990.
- [HMDW91] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. COINS Technical Report 91-47, University of Massachusetts, Amherst, September 1991.
- [HMar] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. *Concurrency and Control: Practice and Experience*, To appear.
- [HMS92] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, October 1992. *ACM SIGPLAN Notices* 27, 10 (October 1992).

- [HMSW00] Richard L. Hudson, J. Eliot B. Moss, Sreenivas Subramoney, and Weldon Washburn. Cycles to recycle: Garbage collection on the IA-64. In *International Symposium on Memory Management*, pages 101–110. ACM, October 2000.
- [Hos91] Antony L. Hosking. Main memory management for persistence, October 1991. Position paper presented at the OOPSLA '91 Workshop on Garbage Collection.
- [Hos95] Antony L. Hosking. *Lightweight Support for Fine-Grained Persistence on Stock Hardware*. PhD thesis, University of Massachusetts at Amherst, MA 01003, February 1995.
- [HWM01] X. Huang, Z. Wang, and K. S. McKinley. Compiling for an Impulse memory controller. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 141–150, Barcelona, Spain, September 2001.
- [JL96] Richard Jones and Rafael D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.
- [KH00] Kin-Soo Kim and Yarsun 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.
- [KMC02] Scott F. Kaplan, Lyle A. McGeoch, and Megan F. Cole. Adaptive caching for demand prepaging. In *Proceedings of The International Symposium on Memory Management*, pages 114–126, June 2002.
- [KSW99] Scott F. Kaplan, Yannis Smaragdakis, and Paul R. Wilson. Trace reduction for virtual memory simulations. In *Measurement and Modeling of Computer Systems*, pages 47–58, 1999.
- [MA90] D. McNamee and K. Armstrong. Extending the Mach external pager interface to accommodate user-level page replacement policies. Technical Report TR-90-09-05, University of Washington, 1990.
- [MGST70] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.
- [MH94] J. Eliot B. Moss and Antony L. Hosking. Expressing object residency optimizations using pointer type annotations. In *Proceedings of the Sixth International Workshop on Persistent Object Systems*, Tarasçon, France, September 1994. Springer-Verlag.
- [Moo84] David Moon. Garbage collection in a large Lisp system. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 235–246, Austin, TX, August 1984.
- [Mos87] J. Eliot B. Moss. Managing stack frames in Smalltalk. In *Proceedings of the ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 229–240, St. Paul, Minnesota, July 1987. *ACM SIGPLAN Notices* 22, 7 (July 1987).
- [Mos89] J. Eliot B. Moss. Addressing large distributed collections of persistent objects: The Mneme project's approach. In Richard Hull, Ron Morrison, and David Stemple, editors, *Proceedings of the Second International Workshop on Database Programming Languages*, pages 269–285, Gleneden Beach, Oregon, June 1989. Morgan Kaufmann. Also available as COINS Technical Report 89-68, University of Massachusetts.

- [Mos90] J. Eliot B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, April 1990.
- [Mos92] J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.
- [MS88] J. Eliot B. Moss and Steven Sinofsky. Managing persistent data with Mneme: Designing a reliable, shared object interface. In K. R. Dittrich, editor, *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, pages 298–316, Bad Münster am Stein-Ebernburg, Federal Republic of Germany, September 1988. *Advances in Object-Oriented Database Systems*, Springer-Verlag, 1988.
- [Nic03] James Niccolai. InfoWorld: Memo reveals doubts at Sun over Java. [http://www.infoworld.com/article/03/02/11/HNjava\\_1.html](http://www.infoworld.com/article/03/02/11/HNjava_1.html), February 2003.
- [PLDM02] Jeffrey Palm, Han Lee, Amer Diwan, and J. Eliot B. Moss. When to use a compilation service. In *LCTES'02-SCOPES'02 Joint Conference on Languages, Compilers and Tools for Embedded Systems and Software and Compilers for Embedded Systems*, pages 194–203, Berlin, Germany, June 2002.
- [SHB<sup>+</sup>02] Darko Stefanović, Matthew Hertz, Stephen M. Blackburn, Kathryn S. McKinley, and J. Eliot B. Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance (MSP 2002)*, Berlin, Germany, June 2002.
- [Sin02] Sharad Singhai. *Data Reorganization for Improving Cache Performance of Object-Oriented Programs*. Ph.d., Department of Computer Science, University of Massachusetts Amherst, February 2002.
- [SKW99] Yannis Smaragdakis, Scott F. Kaplan, and Paul R. Wilson. EELRU: Simple and efficient adaptive page replacement. In *Proceedings of the 1999 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 122–133, June 1999.
- [SMM00] Darko Stefanović, K. S. McKinley, and J. Eliot B. Moss. On models for object lifetime distributions. In *The International Symposium on Memory Management*, pages 137–142. ACM, October 2000.
- [SMS02] Milan N. Stojanovic, Tiffany Elizabeth Mitchell, and Darko Stefanović. Deoxyribozyme-based logic gates. *Journal of the American Chemical Society*, 124:3555 ff., 2002.
- [SRW98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.
- [Ste93] Darko Stefanović. The garbage collection toolkit as an experimentation tool, October 1993. Position paper for OOPSLA '93 Workshop on Memory Management and Garbage Collection.
- [Ste99] Darko Stefanovic. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, 1999.
- [TB98] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):724–767, July 1998.

- [TD96] David Tarditi and Amer Diwan. Measuring the cost of memory management. *Lisp and Symbolic Computation*, 9(4), December 1996.
- [TO01] Guanshan Tong and Michael J. O’Donnell. Leveled garbage collection. *Journal of Functional and Logic Programming*, 2001(5):1–22, May 2001.
- [UWK<sup>+</sup>01] O. S. Unsal, Z. Wang, I. Koren, C. M. Krishna, and C. A. Moritz. On memory behavior of scalars in embedded multimedia systems. In *Workshop on Memory Performance Issues*, Goteborg, Sweden, June 2001.
- [WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116, Kinross, Scotland, September 1995. Springer-Verlag.
- [WKS99] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of The 1999 USENIX Annual Technical Conference*, pages 101–116, June 1999.
- [WLM91a] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective “static-graph” reorganization to improve locality in garbage-collected systems. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 177–191. ACM Press, 1991.
- [WLM91b] Paul R. Wilson, Monica S. Lam, and Thomas G. Moher. Effective “static-graph” reorganization to improve locality in garbage-collected systems. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 26(6), June 1991.
- [WMR01] Zhenlin Wang, Kathryn S. McKinley, and Arnold L. Rosenberg. Improving replacement decisions in set-associative caches. In *Proceedings of MASPLAS’01, The Mid-Atlantic Student Workshop on Programming Languages and Systems*, Hawthorne, NY, April 2001. IBM T.J. Watson Research Center.
- [WMRW02] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, VA, September 2002.