Isla Vista Heap Sizing: Using Feedback to Avoid Paging

Chris Grzegorczyk Sunil Soman Chandra Krintz Rich Wolski

Computer Science Department
University of California, Santa Barbara
{*grze, sunils, ckrintz, rich*}*@cs.ucsb.edu*

Abstract

Managed runtime environments (MREs) employ garbage collection (GC) for automatic memory management. However, GC induces pressure on the virtual memory (VM) manager, since it may touch pages that are not related to the working set of the application. Paging due to GC can significantly hurt performance, even when the application's working set fits into physical memory.

We present a feedback-directed heap resizing mechanism to avoid GC-induced paging, using information from the operating system (OS). We avoid costly GCs when there is physical memory available, and trade off GC for paging when memory is constrained Our mechanism is simple and uses allocation stall events during GC alone to trigger heap resizing, without user participation or OS kernel modification. Our system enables significant performance improvements when real memory is restricted and similar to, or better performance than, the current state-of-the-art MRE, when memory is unconstrained.

1. Introduction

Garbage collection (GC) is a commonly used technique in managed runtime environments (MREs), in order to enable memory safety and improve programmer productivity. However, because the garbage collector must automatically determine how and when to scavenge unused memory *while* the program runs, it introduces execution overhead that can be significant. Thus, optimization techniques are needed to achieve the memory safety and productivity benefits of GC with the minimum possible overhead.

When a garbage collector executes, it must scan the objects that have been allocated by a program to determine which ones are unreachable and can be reclaimed. A large body of research has focused on the performance characteristics associated with different algorithms, and methodologies for implementing GC (e.g. [10, 15, 24, 8, 2, 30, 18, 19, 29, 6, 23]). In our work, we instead focus on the interaction between the garbage collector and the virtual memory system as a potential opportunity for optimization. We observe that given current processor and memory speeds, the cost of paging to disk is significantly higher than that of executing the reclamation algorithm itself, since data must be copied between physical memory and a high-latency disk. Moreover, GC commonly implies maximal use of available virtual memory for the heap, and so GC activity can induce paging *even when the working set of the application fits into real memory*.

Our work explores a new optimization method that uses explicit interactions between the garbage collector and the operating system (OS) virtual memory implementation to avoid GC-induced paging. In particular, our approach is to trigger a collection within the MRE and to perform heap resizing, immediately prior to paging activity that results from a combination of memory pressure and GC activity. We identify "allocation-stall" events in the Linux virtual memory system as an effective harbinger of the onset of GC-induced paging. Our work indicates that substantial performance benefits are achievable by querying the Linux kernel virtual memory data structures to determine when allocation stalls occur (which we do through a dynamically loadable kernel module, thereby avoiding the need to patch and recompile the kernel).

Note that it is also possible to avoid GC-induced paging by sizing the application heap statically [28] via MRE command-line parameters. However, the user must have intimate knowledge of the heap behavior of the application, know the amount of available physical memory in the system, and understand how the OS manages and shares virtual pages. Further, the amount of physical memory available to an application changes dynamically as other applications executing on the system compete for memory pages. Using a static approach, competing applications or user error in heap size selection may result in unnecessary paging as the amount of available physical memory fluctuates. Thus, to be generally effective, we propose an automatically and

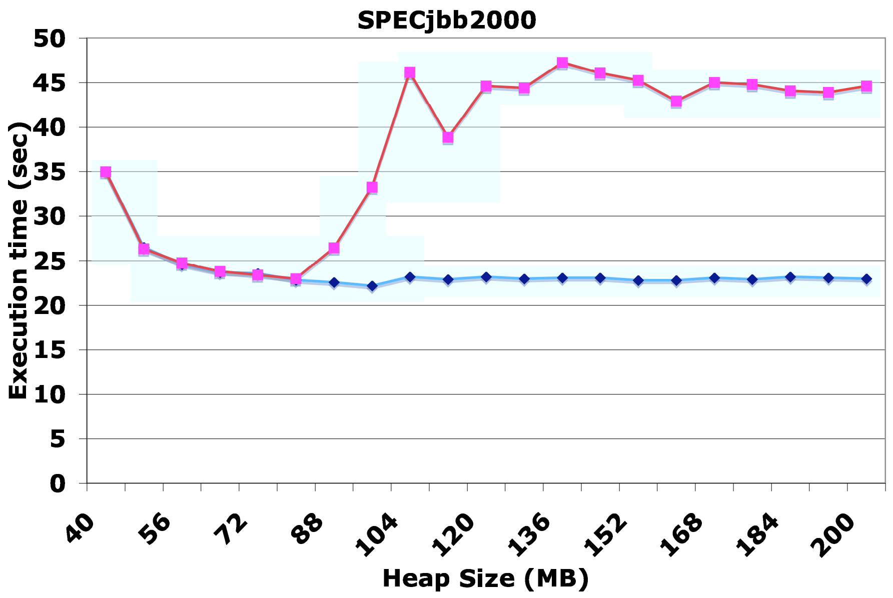

Figure 1. The performance of the SpecJBB2000 benchmark with (squares) and without (diamonds) memory pressure. For the former, we induce memory pressure by and locking 700MB out of 1GB of physical memory. The ungraceful degradation in performance when the benchmark experiences memory pressure is typical of other programs and is caused primarily by paging in the Linux virtual memory system that is induced by GC in the MRE (JikesRVM).

dynamically adaptive technique.

Figure 1 illustrates the effect of GC-induced paging for an example program. The data consists of execution time in seconds (y-axis) versus a heap size (x-axis) specified by the user on the command line. We execute the SpecJBB2000 [25] benchmark on a dedicated system and mlock 700MB out of 896MB of physical memory on the target execution platform to induce memory pressure. We compare the observed performance to execution when no memory pressure is induced. The curve marked with squares shows that when the heap size is set to more than 80MB, program execution performance degrades sharply as the garbage collector induces paging. The curve marked with diamonds, however, shows that execution performance is relatively constant when there is no memory pressure and thus, no paging. The curve clearly exposes the non-linear, and ungraceful performance degradation under memory pressure as a consequence of paging induced by GC.

This GC-induced paging problem has been addressed in prior work by sizing the application heap during program to improve execution performance [4, 11, 13, 33, 34]. However, these approaches require extensive modification to the OS or programs, use complex mechanisms for GC-paging avoidance, are not reproducible or portable without extensive manpower, or introduce a new garbage collector which is virtual memory aware (e.g. [17]).

In this paper, we present a novel and simple feedback-based system relying on virtual memory event statistics (maintained by the OS) to guide the MRE heap sizing policy. We delay GC when memory is available by growing the heap and shrink the heap to trigger GC and to reduce the virtual space used by the GC, when memory is constrained. Our system detects the latter through a light-weight feedback mechanism which distinguishes between the different states of MRE execution (application execution and GC activities): the MRE samples OS events *only at the end of a GC*, minimizing performance impact. Further, the OS event that we monitor is the *allocation stall*, which indicates that no additional free pages are available and that the OS *must swap out pages to disk for the current process (the GC in the MRE)* to allocate physical memory.

By coupling state-awareness and observation of allocation stall events, we can accurately identify when the MRE should resize the heap: Allocation stalls provide a clean signal that is highly correlated with the paging induced by GC activity. Prior work [17, 34, 32, 33] on heap sizing employs page fault and page out events in different ways to detect GC-induced paging. We show, through an analysis of the Linux virtual memory management subsystem implementation that detection based on allocation stalls is more accurate for our system.

We implement the necessary OS profiling and feedback-driven heap resizing optimization mechanism using the commonly available Linux kernel version 2.6.16.9 [1] and the Jikes Research Virtual Machine (JikesRVM) [5] from IBM's T. J. Watson Research Center. Our system is available as a dynamically loaded module for Linux and a JikesRVM patch that is GC-independent [21]. Our empirical results indicate that our system avoids GC-induced paging to enable significant performance gains when physical memory is limited, yet achieves similar (or better) performance levels to the best-performing JikesRVM garbage collector when memory is unconstrained. Moreover, our system is significantly simpler than those reported in previous work: there is no additional burden on the programmer, no OS kernel modifications, and no need for a new GC implementation. Our system also avoids poor performance due to the specification of an inappropriate heap size on the command-line by a JikesRVM user.

In summary, we make the following contributions:

- We define a novel and simple, feedback-driven MRE heap sizing algorithm that uses explicit interactions between the MRE and the OS virtual memory subsystem to avoid GC-induced paging.
- We describe a light-weight sampling mechanism (and its implementation as an Linux kernel module), that captures virtual memory events and associates them with a particular state of execution: program execution or GC.
- We present empirical evidence of the correlation between allocation stalls in the memory manager during GC and impending performance degradation due to paging.
- We implement our system using the popular Linux OS and JikesRVM MRE and make its implemenation freely available [21]. We also empirically evaluate our implementation using a number of community benchmarks. Our extensions are simple and easy-to-use and do not require modification to, and thus potential instablity of, the Linux kernel and JikesRVM garbage collectors.

We next provide an overview of paging events in the Linux virtual memory system, and describe the specific garbage collector and MRE (JikesRVM) that we investigate in this study. Section 3 describes our virtual memory manager profiling and the feedback mechanism we use for dynamic heap sizing. We then present our empirical results, related work, and conclusions.

2. Background

We first review the functionality implemented by the Linux virtual memory manager subsystem, and the JikesRVM GC system that we employ for this work. Although, we focus on these specific technologies, our description is generally representative of Linux/Unix virtual memory behavior and popular generational garbage collection systems.

2.1. Overview of Linux Virtual Memory Management

In the Linux v2.6 kernel, the virtual memory manager (VM) manages a fixed number of physical pages available to applications. As demand for physical memory changes, the VM reclaims currently used pages. To do so, victim pages are identified by scanning the active and inactive lists that are populated with pages that have and have not been recently accessed, respectively. Kernel functions perform the following tasks while the lists are scanned: find pages that can be reclaimed immediately, move pages between the active and inactive list, and write modified (dirty) pages to disk. The latter is termed a *page out*. When a page that is not in memory is referenced by a process, a *page fault* occurs and the VM copies the page from disk to main memory. The latency associated with the transfer of a previously used page either to disk (when a dirty page is written), or from disk (when a page fault needs to be satisfied) is typically 5 or more orders of magnitude larger than processor or memory operations [17].

The VM is invoked to begin reclaiming pages in the following two ways. First, when the number of free pages is low (below a configuration-set threshold), the kernel awakens a dedicated process, called `kswapd`, that starts freeing pages in the background. Second, when the number of free pages drops below a specified absolute minimum level, the virtual memory subsystem does not allow the allocation to progress. Instead, when a process attempts to allocate a page, the kernel logically pauses the allocation process, calls the functions that are normally called by `kswapd`, and then resumes the allocation call path. Such an event is termed an *allocation stall*, as the process of page allocation is stalled while the `kswapd` kernel code reclaims pages.

Prior work [17, 34, 32, 33] makes use of page outs and major page faults as indicators that garbage-collector induced paging will occur. We discuss the choice of which event type to use as a trigger for our algorithm in Section 3.1.

2.2. JikesRVM and Generational/Mark-Sweep Collection

The MRE infrastructure used for this study is the widely used Jikes Research Virtual Machine (JikesRVM) from IBM T. J. Watson Research Center [5], and the modular Memory Management Toolkit (MMTk) [9]. We employ the

Generational/Mark-Sweep collector(GenMS) for this study. GenMS is a widely accepted and highly effective generational GC, and is arguably the best-performing of the available JikesRVM GC systems. Generational collectors divide the heap into multiple, independently collected, regions based on the well-known generational hypothesis that most objects die young. GenMS divides the heap into a *nursery* for young objects, and a *mature*, or old, generation for objects that have survived one or more nursery collections. Full heap collections involve collecting the nursery as well as the mature generation.

The boundary between the nursery and the mature space varies dynamically (using an Appel-style nursery [6]). Initially, before any garbage collection has been triggered, the mature generation is empty and the nursery occupies half the heap. The second half is the *copy reserve* and is necessary to handle worst-case liveness behavior during the next collection. The system triggers a nursery collection when the nursery is full, which is an *en masse* promotion of live objects to the mature generation, which causes the mature space to grow. The system resizes the nursery (active space) to occupy half of the space remaining. A full heap garbage collection involves promotion of live nursery objects, followed by collection of the mature generation. GenMS performs this mature space collection using mark-sweep (MS) [23].

The system we propose and implement herein is independent of the GC since we only manipulate the heap sizing mechanism within JikesRVM/MMTk (which we describe next). In this paper, we only investigate the performance of our system when we use GenMS for collection.

Since a nursery collection is significantly less expensive than a full heap collection (many objects die young and the collector performs less work for the former), our goal is to delay full heap collection by growing the heap. However, a heap which overflows the available physical memory will necessarily degrade performance due to GC-induced paging. For a generational heap, the common reference behavior as the heap grows will result in the mature space being swapped out first, followed by the nursery's copy reserve space, and then the nursery's active space. With the mature space in swap, each page will be faulted in at least once during a full heap collection. Major page faults for pages will be caused by nursery collection when the copy reserve has been swapped to disk, and by the application allocating objects when the active space resides on disk.

3. Feedback-driven Heap Sizing

Given the interaction between GC and VM paging, the goal of our work is to avoid full-heap GCs as much as possible when physical memory is *unconstrained*, but to trigger heap resizing (inducing GC) to avoid paging when memory becomes scarce. Aggressively heap growth enables the nursery to grow quickly, and memory management to occur via inexpensive nursery collections. However, overflowing physical memory will result in paging and must be detected. In combination with the high cost of paging relative to GC (even full heap collections), we must shrink the heap and keep the heap in memory (i.e. trade-off paging for GC).

A significant body of prior work employs heap sizing techniques guided by OS events to reduce, or eliminate GC-induced paging [17, 4, 11, 13, 33, 34]. We detail and contrast this work to our own in Section 5. In summary, our system is unique in the type of feedback used to guide (trigger) resizing, and when our policy is applied. More so, our solution is very simple and does not require Linux kernel modification (we contribute a kernel module only – and do so only for efficiency reasons), application modification, offline profiling, or participation by the user. In addition, we make only minimal and modular modifications to the MRE ($<$ 200 lines of code). In the subsections that follow, we describe the trigger and sample collection mechanisms and then detail our heap resizing algorithm.

3.1. State-Aware Sampling and Trigger Selection

We have identified three candidate VM events for indicating when memory pressure will induce paging by the GC: page outs, page faults, and allocation stalls. All three values are recorded by the Linux VM using simple counters, thus the cost of accessing each of them is the same and low. However, in a set of exploratory experiments using each method, allocation stalls outperform the other two choices. Analyzing the Linux virtual memory subsystem explains our conclusion.

Recall from Section 2 that Linux uses a separate *kswapd* process to manage memory asynchronously. Periodically, *kswapd* will initiate page outs in a system that is not experiencing serious memory pressure. The VM's management of pages without regard for process ownership combined with the asynchronous nature of paging to disk implies two problems in using pageouts. First, a GC during which pageouts occur *does not* imply that heap pages have been swapped out, nor that the heap is inducing memory pressure. Since GC will trace all heap pages during a full collection, the reference information used by the VM to select pages for eviction will be tainted. As a result, heap pages will be more likely to remain resident implying that the pageouts recorded would spurious – from other applications. Second, as the act of sending a page to disk is not performed synchronously, no guarantees exist that observed pageouts resulted from or even occured during MRE execution.

Then, examining major page faults in the same light reveals a similar problem: the source and timing of the culprit responsible are unclear. Major faults, while occuring synchronously, can be the result of an asynchronous page-out, one not related to MRE activities. Instead the VM may choose to evict heap pages in response to any pressure, regardless of the source.

On the other hand, an allocation stall *only* occurs when memory pressure is severe and the kernel must engage in memory page management immediately. More importantly, because allocation stalls are serviced in the kernel process context of the currently executing application, they unambiguously identify the process responsible for inducing memory pressure. Following the passive efforts of *kswapd* to shrink active working sets to fit into main memory, an allocation stall corresponds directly with pages being sent to disc synchronously. Combined with the observation that pages are only allocated when the MRE is in a GC phase, allocation stalls during garbage collection are more strongly correlated with the heap overflowing real memory.

To confirm this hypothesis, we count the number of GCs during which one of the three events occurs while running SPECjbb2005 for a range of heap sizes (40MB-256MB, with 700MB mlocked): Across the heap sizes small enough to be memory resident, spurious page faults occurred every time, pageouts appeared less frequently, but allocstalls almost never did. For the remaining heap sizes, an interesting result is evident: the number of garbage collections which observe a page fault, pageout, or allocation stall satisfy the ration 4:2:1, respectively. That is, for each GC which induces an allocation stall, there are two measured pageouts, and four page faults. The ultimate indicator of allocation stall's utility at indicating memory pressure is revealed by inspecting the sequence of event observations during garbage collection for a single execution (with a heap that overflows memory): GCs which observe allocstalls precede those with page faults.

Thus, by sampling the kernel's allocation stall counter immediately before and after each garbage collection event, our system correctly identifies memory pressure caused by garbage collection activity in a specific process. We term this sampling process "state-aware" feedback, since the recording of the operating system's virtual memory activity pertains only to the state of the program that is relevant (in a GC or not).

Notice that neither page outs nor page faults permit this level of specificity without modification to the Linux kernel's accounting procedures, potentially destabalizing the VM. Page outs may occur asynchronously as well as synchronously (i.e. in the current process context in response to an allocation stall). To use them as effectively, the kernel would need to differentiate (at a very minimum) between the two different types of page out events. Page faults, on the other hand are executed synchronously so that activity by a specific garbage collector can be identified. However, a page fault during garbage collection occurs in response to memory pressure that may or may not have been induced by the MRE itself. Put another way, a page fault during garbage collection is a consequence of a page out *that has already occurred*, and may have been caused by the activity of another process. In contrast, when an allocation stall occurs during garbage collection, it must be the activity of the garbage collector that is responsible for the memory shortfall. Thus, allocation stall events provide a more accurate indication of paging that can be avoided by heap resizing.

3.2. Feedback System Design

The feedback mechanism enables the MRE to interact with the OS to obtain allocation stall information in a state-aware manner. The MRE communicated with the OS before and after each GC (nursery and full), to obtain allocation stalls. The implementation consists of two parts: a kernel module, and an interface for the MMTk. The interaction is enacted through a file in the /proc filesystem provided by the module. Several characteristics are worth noting. First, the VM data structures in the kernel are touched only twice per sample (at start and finish) by code in the module. No changes are needed in the critical VM subsystem. As a result, the VM behaves in the standard and tested way – ensuring that our system does not influence stability. Second, the module exposes a synchronous single file read-only interface, encapsulating all of the necessary logic. The MRE portion of the implementation is accomplished using only low level read calls. As a result, the synchronization provided by the interface guarantees correct state-aware sample reporting, and minimizes the need for changes to the sensitive GC subsystem. Third, all the MRE code resides in an MMTk source tree that is external to the JikesRVM source. We, thus, provide an uniform MMTk interface that can be used by other GC-based runtimes that use the MMTk, i.e., our implementation is independent of the execution environment (Java Virtual Machine).

In our work, we have explored other ways to access information about memory pressure. In particular, we examined using kprobes and the information available via existing /proc entries. Kprobes require the use of a module and problematic to use in the memory manager path (e.g. probing that causes a fault that causes probing). While the information we seek can be obtained via /proc, by embedding the profiling logic inside of our kernel module we ensure the correctness of our measurements and avoid the prohibitive performance impact of having the rest of the system in user space – influencing the MRE/GC. In essence, our module extends the functionality of various /proc entries by providing significantly more efficient state management. Implementing state-aware profiling using existing /proc entries is possible, but would imply managing state in user space where the synchronization required to guarantee accurate measurements may negatively impact performance and stability. To achieve the performance needed, a kernel module is necessary to modify the standard /proc functionality.

3.3. Isla Vista Heap Sizing

The HeapGrowthManager is responsible for determining the heap sizing policy in the MMTk. The existing approach to resizing is to use the execution time spent in GC to determine the heap sizing ratio, after every full heap collection. We replace this algorithm with our own, which we call *Isla Vista (IV) Heap Sizing*, that resizes the heap according to the allocation stall activity that is captured by our state-aware sampling system. IV heap sizing makes a decision about sizing at *every* GC, i.e., both nursery and full heap collection, as opposed to only at full heap collection in the original system.

To implement IV heap sizing, we draw inspiration from the TCP Reno congestion control [22, 12, 27, 3]. TCP Reno uses loss events to estimate available bandwidth. The algorithm manages a *congestion window* that determines the number of packets that can be "in-flight" at one time. When a loss event occurs, the system halves the window size. Otherwise, it is allowed to grow linearly. The essential property of this *additive increase/multiplicative decrease* scheme is to identify a window size that enables convergence to a fair allocation scenario [12].

We implement a similar strategy in the IV heap sizing algorithm. We grow the heap linearly when there are no allocation stalls and shrink the heap aggressively and slow the growth factor for successive heap growth decisions, when we detect allocation stalls during GC. This process attempts to converge to a heap size that effectively balances the trade-off between paging and GC cost.

We refer to the total size (used and unused) of the nursery and the mature space as $heap_{limit}$. IV heap sizing increases and decreases $heap_{limit}$ to grow and shrink the heap, respectively. When the nursery reaches $heap_{limit}$, the system performs a nursery collection. A collection free of allocation stalls indicates that the system has walked through the entire nursery area without inducing page outs due to memory pressure. We record the current $heap_{limit}$ (before resizing) as $lastHeap_{limit}$ and the size of the reserved space following this collection, in case we need it to shrink the heap during subsequent collections. The reserved space following collection is the size of the live data, i.e., the size of the mature space. We record this value as $lastHeap_{reserved}$ for later use. We grow the heap by increasing $heap_{limit}$ which we explain below.

When a GC experiences an allocation stall, we shrink the heap and the growth factor. To shrink the heap, we set $heap_{limit}$ to be $lastHeap_{limit}$ minus $lastHeap_{reserved}$, i.e., the size of the previous nursery. We do this so that, in the worst case (i.e. when all of the previous nursery was live and promoted to the mature space), we cut the nursery size in half. In all cases, the new nursery size is between 1/2 and 1 times the previous nursery. We essentially "fit" the mature space into the number of pages used by the last successful nursery traversal (for which there were no allocation stalls during collection). We never shrink the heap below $lastHeap_{reserved}$ in the case where multiple shrink events are initiated successively (due to increasing memory pressure).

When we grow the heap, we increase $heap_{limit}$ by an additive factor. We compute this factor, called $step$, as $step_{base} \times ratio_{grow}$, where $step_{base}$ is originally the maximum heap size specified by the user (to impose an upper bound on growth to start with) minus $heap_{limit}$. $ratio_{grow}$ is a value between 0 and 1; we use the value 0.02 in this paper. When we shrink the heap, we also shrink $step_{base}$ to slow subsequent growth events. In this case, we update $step_{base}$ to be $step_{base}$ - $step_{base} \times ratio_{shrink}$, where $ratio_{shrink}$ is a value between 0 and 1; we use the value 0.75 for $ratio_{shrink}$.

We have selected values for $ratio_{grow}$ and $ratio_{shrink}$ based on our experience with Java benchmarks and their memory behavior, e.g., the frequency of their collections. We set $ratio_{grow}$ to be small so that we maximize the heap within a small number of collections. We set $ratio_{shrink}$ to be closer to 1 (than say 0.5) so that we perform shrinking often enough to improve performance without being too aggressive.

On experimenting with various values for $step_{base}$, we found that our system is effective when this value ranges from 0.02 to 0.05. Varying $step_{base}$ within this range does not significantly impact performance. We found that on setting $step_{base}$ to a value outside this range, JikesRVM experienced failures, most likely due to a bug in the baseline (unmodified) system itself. Consequently, we were unable to directly measure the impact of setting $step_{base}$ to an arbitrary value. However, the values that we have selected are very effective for all of the benchmarks we have considered.

4. Results

In this section, we present an empirical evaluation of our system. We first describe our experimental methodology and then present our results.

4.1. Experimental Methodology

Our implementation of Isla Vista heap sizing is a kernel module for the vanilla Linux kernel v2.6.16.9 and source module for JikesRVM/MMTk from the CVS head dated 7/19/06. To capture the detailed timing and event count-

Table 1. Benchmark Descriptions.

Benchmark	Description	Mlocked Size	Input
hsqldb	JDBC-like in-memory benchmark, executing a number of transactions against a model of a banking application	700	-s default
pmd	Analyzes a set of Java classes for a range of source code problems	640	-s default
bloat	Performs a number of optimizations and analysis on Java bytecode files	640	-s default
Db	SPECJvm98 database access program	736	-s 100
Javac	SPECJvm98 Java to bytecode compiler	736	-s 100
SPECjbb2000	Transaction processing application (we execute a fixed number of transactions)	700	-t 250000 -w 1

ing information necessary to fully analyze IV, we have implemented a separate profiling utility, independent of our state-aware heap sizing system, that enables collection of low level OS events. This profiler requires modification to the kernel (and thus recompilation) for instrumentation of the fault path of the kernel, and to enable our precise measurements of the overhead due to different types of faults; some of which we include in this paper. Note that IV itself does not require modification to the kernel – we implement the feedback mechanism (state-aware sampling) using a dynamically loadable kernel module. To avoid introducing spurious overhead, we collect results using a separate profiling process (which interacts with JikesRVM) whose entire memory allocation is locked (using mlock) prior to execution. We lock 300MB of memory for the collection of profile data in all of our experiments. Moreover, we use this profiling process to induce memory pressure artificially by increasing the amount of memory it locks.

For all of our constrained memory experiments, we induce memory pressure in this way (similar to that used in [17]) so that we may make repeated empirical experiments in a timely manner. The systems we have at our disposal have sufficient memory so that for the benchmarks to encounter memory pressure “naturally”, we would need to run problem instances with execution times that would preclude the repeated execution of each instance that is necessary for statistical rigor. Due to the balancing done by the Linux memory manager between various memory pools (file buffers, slab allocation, process mapped memory), computing the exact amount of memory available to processes at any point in time is difficult. For expediency, we report the amount of physical memory available and the amount of memory we have intentionally mlocked to preclude it from being committed during virtual memory activity by our test system.

We perform our experiments using the benchmarks and inputs described in Table 1. They are from the SPECjvm98 [26], SPECjbb2000 [25] (using a fixed number of warehouses), and Dacapo [14, 16] suites. We selected the benchmarks from these suites that exhibit some measurable paging behavior within our experimental setting. Detailed statistics for these benchmarks are available in [16]. In column three of the table, we indicated the amount of memory that we mlock for each program (which ranges from 640MB to 736MB). This value is the total mlock value including the 300MB that we employ for result collection.

We generate our results using a 3.2GHz Pentium Xeon machine (16KB L1 cache and 1MB L2 cache) with 896MB of addressable memory (HIGHMEM is disabled in the kernel) and 5GB of swap. Unless otherwise stated, all of the values that we present are the average of 10 runs – we include the range of one standard deviation above and below this value in the graphs. We follow [17] and run each benchmark twice. We use the optimizing compiler during the first run after which, we disable it and perform a full heap collection. We then collect our performance results using the second run during which no re-compilation is performed. We then execute this process for each benchmark 10 times and report the average across these 10 runs with error bars that represent one standard deviation on either side of the average.

We first show the impact memory pressure has on application execution for our benchmarks. We then show the performance improvement enabled by our heap sizing mechanism and compare it to performance under no memory pressure. Finally, we compare the performance of IV to the baseline system with no memory pressure.

4.2. Evaluation

We begin by measuring the effect of memory pressure on the performance. In Figure 2, we present degradation versus

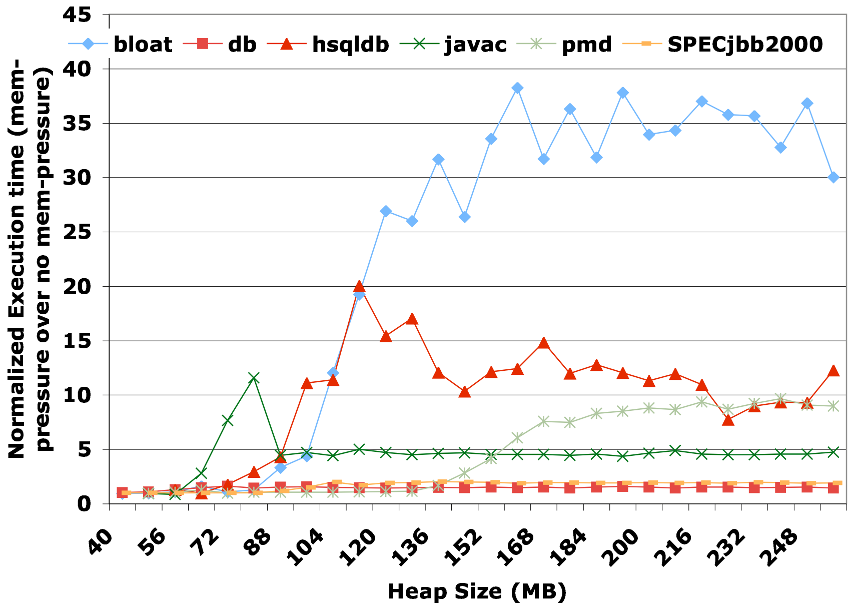

Figure 2. Execution time of the benchmarks under memory pressure, normalized to their execution time under no memory pressure. We induce memory pressure artificially as we describe in Section 4.1. Paging imposes performance degradations that range from a few percentage points to 38X.

maximum heap size as specified by the "-Xmx" commandline parameter to JikesRVM. Each data point is the execution time of the particular benchmark under memory pressure, divided by the corresponding execution time of that benchmark under no memory pressure. The worst-case degradation due to memory pressure ranges from a factor of 1.5 to a factor of 38 across benchmarks. The data in this graph shows both that the performance degradation due to memory pressure is significant across a range of user-specified maximum heap sizes, and that it is highly benchmark specific.

We hypothesize that the degradation shown in these observations under memory pressure is due to paging, and also that the cost of paging is significantly higher than that of GC. We have verified that this is the case by comparing paging and GC overhead under these scenarios. We summarize this data later in this section. However, if this hypothesis is true, then our methodology which avoids page faults at the expense of additional collections should show substantial performance improvement under memory pressure.

Figure 3 shows these results. The graphs show for benchmark and a range of specified maximum heap sizes, the execution time in seconds for the baseline (Base) system which uses the default JikesRVM heap sizing algorithm and for IV heap sizing. For both scenarios (Base and IV), we present results for when we induce memory pressure (MemPressure) and when we do not (NoPressure).

For all benchmarks and heap sizes, IV significantly outperforms the baseline heap growth manager when under the same memory pressure. On average, performance IV improves performance over the baseline by factors ranging from 2 to 10. Moreover, IV performance in the presence of memory pressure is similar to the baseline system's performance without memory pressure (Base-NoPressure). IV execution under memory pressure is always within a factor of 2 of the baseline system without memory pressure, with several cases being only slightly slower. Finally, without memory pressure, IV in some cases achieves better performance than the baseline. This improvement is a consequence of the ability of our heap growth mechanism to reach a larger heap size, faster, i.e. more aggressively, in the absence of memory pressure.

The memory pressure data also exhibits some interesting performance characteristics and anomalies. Most notable (and perhaps counter-intuitive), is the similar shapes in the curves. Each benchmark exhibits a similar performance trajectory (although at different heap sizes): performance is efficient and stable for small heap sizes, it then degrades quickly (due to a combination of paging and GC) and then "plateaus" at a very low performance level. This behavior emphasizes the importance and potential of the intelligent heap sizing of IV to avoid ungraceful degradation and the poor-performance of the plateau.

Other performance characteristics under memory pres-

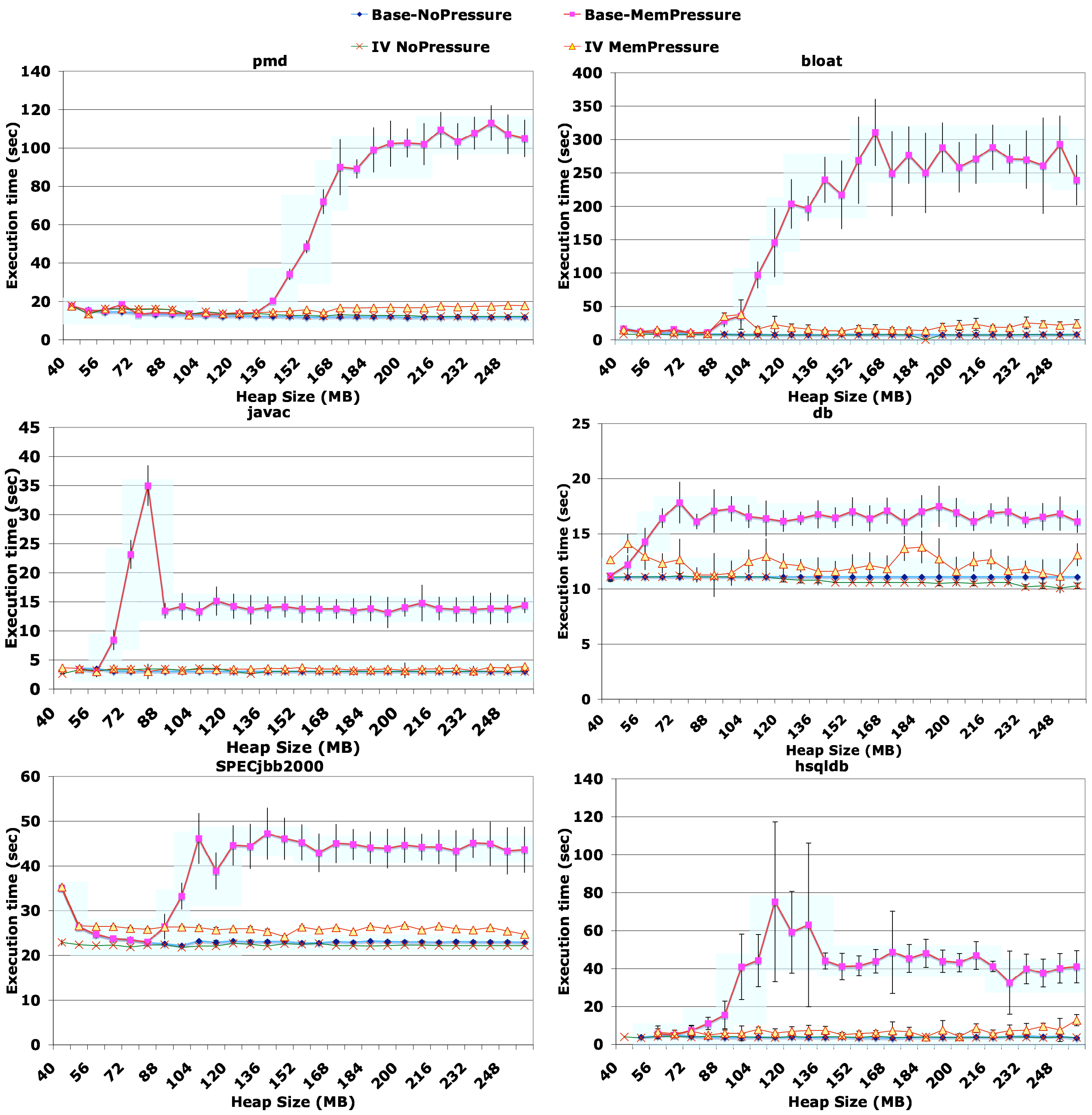

Figure 3. Performance versus maximum heap size. The graphs show, for each benchmark, the execution time in seconds when for the baseline (Base) system which uses the default JikesRVM heap sizing algorithm, and for Isla Vista heap sizing. For both (Base and IV), we present results for when we induce memory pressure (MemPressure) and when we do not (NoPressure). In all cases, IV enables performance behavior similar to that when under no memory pressure, a dramatic improvement over the baseline system.

sure that we observe in this data are benchmark-specific or similar across certain benchmarks. With 640MB locked, pmd and bloat begin to experience performance degradation at different heap sizes (128MB and 80MB, respectively). With 736MB locked, db and javac also behave differently with db suffering degradations early on, for heaps as small as 40MB; javac starts experiencing degradation at around 56MB. With 700MB locked, the performance of hsqldb and SPECjbb2000 starts to degrade around the same heap size, 72MB. However, there is a marked difference in the impact the next increment in heap size: SPECjbb2000 jumps dramatically while hsqldb degrades more gracefully. hsqldb and javac having a large spike followed by a lower smooth plateau as the heap sizes increase. For both these programs, there are no full heap collections triggered when the heap size reaches the "plateau" region. Consequently, there is a sudden drop in execution times.

We next present more detailed analysis of the performance of IV versus the baseline. Table 2 presents GC and page faults metrics. The table shows, for each benchmark and system configuration (IV or Base), the count of and time spent in nursery collections, the count and time spent in full heap collections, and the count and time for major page faults. In all cases, we present time in seconds.

The data shows how IV heap sizing trades off GC for VM paging. IV incurs more GCs compared to the baseline system, and induces significantly less paging compared to the baseline. For example, if we consider SpecJBB2000, the data shows that IV heap sizing performs 283 nursery, and 21 major collections, compared to 51 nursery, and 26 major collections for the baseline system. This amounts to 227 extra garbage collections (nursery and full) in case of IV heap sizing. Yet, we induce 0 page faults; the baseline spends almost 13 seconds page faulting. The fraction of GCs that are overhead in case of IV heap sizing can be computed as the total number of extra GCs for IV divided by the total number of GCs for IV. In this example (SpecJBB), this fraction is 0.75 (227/304). This fraction multiplied by the total time per GC for IV (nursery + full heap, i.e. 2.010 + 2.306), gives us the extra GC time that we add, i.e. 0.75 * 4.316, or 3.24 seconds in this example. That is, in this example, we trade 3.24 seconds spent in extra garbage collection for almost 13 seconds saved by not incurring page faults.

In Table 3, we examine the performance of IV relative to the Baseline in greater detail. The table is organized as three columns, each of which is subdivided to show two cases (80MB and 240MB specified maximum heap size). Each of the three columns illustrates the performance of IV as compared to the Baseline under the relevant set of memory pressure conditions. Each value is a ratio of execution times, baseline divided by IV. In the first major column, we show the ratio of the baseline execution time to the execution time of IV under memory pressure. If the maximum heap size specified is large enough to allow IV to expand the heap sufficiently, (e.g. the 240 MB case) then IV outperforms the baseline across benchmarks in the presence of memory pressure. Notice that for pmd and SPECjbb2000 there is a slight performance degradation relative to baseline. (Note that this degradation is not visible in Figure 3 since it represents approximately 0.5% of the maximum value shown on the graph).

In the second major column, we show the ratio of baseline times to IV times, when the baseline experiences no memory pressure, but IV does. Values in this column detail how close IV is able to come to completely alleviating the performance degradation introduced by memory pressure entirely (i.e. how close IV can come to the case for the baseline with no memory pressure). Here the results are mixed. For several benchmarks and maximum heap size combinations, IV is able to closely approximate pressureless execution (achieving 90% or better). For a few benchmarks, however, while it improves performance (as shown in Figure 3) the percentage improvement does not achieve the same speed as the baseline without memory pressure.

In the third major column, we compare IV to the baseline when neither experiences memory pressure. In this case, undersizing the maximum allowable heap causes IV to suffer relative to the baseline due to additional collections it induces. However, again when the maximum heap size allows IV sufficient flexibility, it achieves similar performance to the baseline.

The data in this table suggests that by specifying a large maximum heap size (e.g. 240MB) and then allowing IV to dynamically size the heap it will achieve the same or better performance to the baseline in both with and without memory pressure. When no pressure exists, IV will match or outperform the Baseline (with the exception of pmd). When pressure exists, using IV to size the heap will always improve performance.

5. Related Work

Much prior work has investigated the interaction between operating system virtual memory (VM) paging behavior and garbage collection (GC) in an MRE. We identify the work most related to our own and articulate the differences here.

The works most similar to our own are the bookmarking garbage collector [17] and specialized VM support within Linux for GC runtimes in a system called CRAMM [32]. In [17], the authors propose a new garbage collector, called the Bookmarking Collector (BC), and extensions to the Linux kernel which enable interaction between the MRE and OS to guide heap sizing decisions. BC maintains a pool of free pages and grows the heap until there are none left. BC records summary information (called bookmarks)

Table 2. The tradeoff between more garbage collections and paging with memory pressure.

Benchmark	Heap Sizing	Nursery GC		Full Heap GC		Major Faults	
		Count	Time (sec)	Count	Time (sec)	Count	Time (sec)
bloat	IV	256	3.025	30	4.053	2753	1.378
	Base	106	33.175	96	48.267	61515	101.745
db	IV	14	0.054	0	0.000	84	0.984
	Base	18	0.271	10	4.832	2920	4.613
hsqldb	IV	107	0.523	2	0.273	123	0.203
	Base	106	8.497	26	12.652	14537	17.624
javac	IV	33	0.420	1	0.172	56	0.000
	Base	29	10.405	0	0.000	4304	5.208
pmd	IV	352	3.261	27	3.463	237	0.000
	Base	127	42.831	60	30.277	33405	46.840
SPECjbb2000	IV	283	2.010	21	2.306	0	0.000
	Base	51	6.296	26	12.733	10059	12.983

Table 3. Performance comparison of IV and the baseline under different scenarios.

Benchmark	Base-MemPressure / IV MemPressure		Base-NoPressure / IV MemPressure		Base-NoPressure / IV NoPressure	
	80MB	240MB	80MB	240MB	80MB	240MB
hsqldb	2.107	3.837	0.713	0.410	0.850	1.142
pmd	0.869	6.484	0.812	0.671	0.822	0.966
bloat	2.167	10.986	0.916	0.335	1.146	1.051
db	1.430	1.446	0.984	0.969	1.000	1.078
javac	11.566	3.744	0.999	0.818	0.882	1.000
SPECjbb2000	0.889	1.714	0.882	0.876	1.023	1.036

about evicted pages to avoid paging during GC, i.e., to perform GC only in available physical memory. In addition, BC guides eviction decisions that are made by the operating system, i.e., the collector is tightly coupled with the underlying OS implementation.

The authors of [33, 32] extend the Linux VM manager to reduce page faults caused by GC for garbage-collected MREs – independent of the GC itself. The system, CRAMM, computes working-set size information on a per-process basis and couples the information with minor page fault events which CRAMM induces via intelligent page write-protection. CRAMM uses the data to parameterize an analytical model to guide heap sizing decisions made by the OS. For each process that uses GC, CRAMM sets (and adjusts) the heap to be as large as possible without inducing page faults.

Our work has the same goal: to eliminate VM paging overhead due to GC. Like CRAMM, we improve performance using, and independent of, existing garbage collectors. However, we are distinct in that we provide a system design which is effective, yet *very simple, easy to use and reimplement*. IV is designed to preserve the integrity of the fundamental components of the OS and MRE (VM and GC algorithms), avoiding potential destabalization. In doing so, we avoid modifying the kernel at all and introduce only minor changes to the MMTk. Prior systems are highly complex in requiring expert ability to install, a new GC ([17]) that is tightly coupled with the OS VM system, or significant modifications to fundamental kernel subsystems (memory manager) and, thus, recompilation of the kernel boot image. Despite the simplicity of our design, IV offers dramatic performance improvements when the system MRE experiences memory pressure and introduces little overhead when there is no memory pressure.

Zhang et al in [34] use program phase behavior to guide heap sizing. The authors identify the phases of execution exhibited in each program offline and modify the program prior to execution to mark phase boundaries. They explicitly check the number of page faults during garbage collection and the current heap size after a fixed number of phases execute. If the number of page faults during these intervals exceeds a certain threshold, the system explicitly invokes a GC. After a GC, the heap is sized by performing a binary search over a range of heap sizes using a fixed step size. The authors define their step arbitrarily to be 10 and their page fault threshold empirically to be 10. On the other hand, IV uses only on-line feedback from the OS after GC to guide heap sizing and requires no modification to the program. Moreover, we find that allocation stall events are better predictors of GC paging behavior than page faults for our system. We are currently investigating whether additional information about program phases can help guide heap sizing decisions.

Xian et al [31] observe that some commercial virtual machines like the Hotspot VM do not take into account physical memory availability when resizing the heap. The heap size often grows larger than the physical memory, which causes performance degradation due to a large number of page faults. They modify the existing heap sizing mechanism in Hotspot to grow the heap by a one percentage value during collection when the heap size is 75% of available physical memory, and by another percentage when it is over 75%. They experiment with different percentage values and observe that they achieve the best performance when the heap expands by 10% until it is 75% of the physical memory, and by 5% afterward. More importantly, their observations show that the best thresholds are application-specific and that conservative heap growth can significantly degrade performance. The authors of this work do not consider shrinking the heap. We exploit these observations in our system to use online feedback and to aggressively grow the heap when we predict that paging will not occur (no allocation stalls during GC). However, we employ allocation stalls to guide resizing. Moreover, we find shrinking the heap is vital for enabling high performance when memory becomes constrained.

Alonso et al [4] uses coarse-grained information from the Linux VM manager via the UNIX utility `vmstat` to guide decisions to shrink the heap when the memory pressure is high. The authors do not perform heap expansion. Brecht et al [11] resize the heap using rules based on application-specific, static memory sizes. Their system allocates memory (according to this static size) to the application upon invocation. Cooper et al [13] use a user specified parameter, called the memory use target, to size the heap for an Appel-style GC. They attempt to adjust the heap to make full use of the available memory, if it matches the user specified target. Other resizing techniques are performed in some commercial Java virtual machines, such as JRocket and HotSpot [7, 20]. These JVMs resize their heap using command line and other statically defined parameters. The primary limitations of these approaches are their inability to adapt to changes in program behavior and memory availability and the significant burden they place on the programmer to identify the "right" heap size for every hardware/OS/application combination. We preclude programmer participation and enable high performance automatically and simply by dynamically adjusting the heap size in the MRE according to paging behavior that we predict using allocation stall events from the OS.

6. Conclusions and Future Work

Using allocation stall event statistics recorded in the Linux virtual memory subsystem as a trigger, Isla Vista dynamically resizes the JikesRVM/MMTk heap to avoid GC-induced paging. Coupled with a linear-growth and exponential reduction resizing algorithm (in the spirit of TCP Reno), IV offers dramatic performance improvements over the unmodified GenMS collector when memory pressure or user misconfiguration causes paging during garbage collection. More so, when adverse conditions do not occur, IV adds relatively small overhead for some benchmarks, and improves performance for others.

IV achieves these results while requiring relatively few non-intrusive modifications to the standard, publicly available JikesRVM and Linux releases. Our system retrieves feedback via a dynamically loadable kernel module, thereby preserving the integrity of the critical memory management subsystem in the kernel and avoiding inconvenient kernel patching followed by recompilation. The changes necessary to JikesRVM are confined to a single additional module that interacts with the kernel module to adjust the heap *limit*.

As part of our future work, we plan to investigate how IV reacts to dynamically changing external memory pressure, due to other applications that might be competing with the virtual machine for resources.

In addition, we plan to investigate how IV's performance gains compare to previously reported systems, which require substantial modification to the kernel, the garbage collector, or both. Such a comparison will detail the execution "cost" of using a non-intrusive approach (with the concomitant benefit of being able to rely on stable, often-tested kernel software) versus a highly specialized approach that requires extensive re-engineering of the system. In this work, we focus solely on the empirical investigation of the performance gains that are possible using a non-intrusive approach versus the unmodified standards: we find that such gains are substantial.

More generally, we plan to evaluate our heap sizing mechanism for other collectors and virtual machines. We are also interested in the use of our technique for other operating systems and architectures. Note that we do not require any specific information about the GC to be used, such as the size of the nursery. We believe that our mechanism can be used with a wide variety of collectors, generational and non-generational.

References

[1] The Linux Kernel Archives. `http://www.kernel.org`.

[2] A. Aiken and D. Gay. Memory management with explicit regions. In *Conference on Programming Language Design and Implementation*, May 1998.

[3] M. Allman, V. Paxon, and W. R. Stevens. Rfc 2581: Tcp congestion control, Apr. 1999. `http://www.faqs.org/rfcs/rfc2581.html`.

[4] R. Alonso and A. W. Appel. An Advisor for Flexible Working Sets. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1990.

[5] 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, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–221, 2000.

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

[7] Technical white paper - BEA Weblogic JRockit: Java for the Enterprise. `http://www.bea.com/content/news_events/white_papers/BEA_JRockit_wp.pdf`.

[8] S. Blackburn, J. Moss, K. McKinley, and D. Stephanovic. Pretenuring for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Tampa, FL, Oct 2001.

[9] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java With MMTk. In *International Conference on Software Engineering (ICSE)*, May 2004.

[10] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *Conference on Programming Language Design and Implementation*, June 2002.

[11] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce execution time of Java applications. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, Nov. 2001.

[12] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Netw. ISDN Syst.*, 17(1):1–14, 1989.

[13] E. Cooper, S. Nettles, and I. Subramanian. Improving the Performance of SML Garbage Collection using Application-Specific Virtual Memory Management. In *ACM Conference on LISP and Functional Programming*, June 1992.

[14] The Dacapo Benchmark Suite, version beta050224. `http://www-ali.cs.umass.edu/DaCapo/gcbm.html`.

[15] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-First Garbage Collection. In *International Symposium on Memory Management (ISMM)*, Oct. 2004.

[16] S. M. B. et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2006.

[17] M. Hertz, Y. Feng, and E. D. Berger. Garbage Collection Without Paging. In *Conference on Programming Language Design and Implementation (PLDI)*, June 2005.

[18] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based Garbage Collection. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 359–373, Oct. 2003.

[19] R. L. Hudson, R. Morrison, J. E. B. Moss, and D. S. Munro. Garbage Collecting The World: One Car At A Time. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 1997.

[20] S. M. Inc. The Java Hotspot Virtual Machine White Paper. `http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_%4_30_01.html`.

[21] IV Heap Sizing Source. `http://www.cs.ucsb.edu/~grze/iv`.

[22] V. Jacobson. Congestion avoidance and control. In *SIGCOMM*, pages 314–329, 1988.

[23] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996.

[24] S. Soman, C. Krintz, and D. F. Bacon. Dynamic Selection of Application-Specific Garbage Collectors. In *International Symposium on Memory Management (ISMM)*, Oct. 2004.

[25] SPECjbb2000. `http://www.spec.org/jbb2000`.

[26] SpecJVM'98 Benchmarks. `http://www.spec.org/osg/jvm98`.

[27] W. R. Stevens. Rfc 2001: Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms, Jan. 1997. `http://www.faqs.org/rfcs/rfc2001.html`.

[28] Sun microsystems white paper: Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine . `http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html`.

[29] D. Ungar. Generation scavenging: A non-disruptive high performance storage recalamation algorithm. In *Software Engineering Symposium on Practical Software Development Environments*, Apr 1992.

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

[31] F. Xian, W. Srisa-an, and H. Jiang. Investigating the Throughput Degradation Behavior of Java Application Servers: A View from Inside a Virtual Machine. In *International Conference on Principles and Practices of Programming In Java*, Aug. 2006.

[32] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual Memory Support for Garbage-Collected Applications. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, Nov. 2006.

[33] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic Heap Sizing: Taking Real Memory Into Account. In *International Symposium on Memory Management (ISMM)*, Oct. 2004.

[34] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara. Program-level Adaptive Memory Management. In *International Symposium on Memory Management (ISMM)*, June 2006.