

Waste Not, Want Not

Resource-based Garbage Collection in a Shared Environment

Matthew Hertz Stephen Kane
Elizabeth Keudel

Department of Computer Science
Canisius College
Buffalo, NY 14208

{hertz,m,kane8,keudele}@canisius.edu

Tongxin Bai Chen Ding
Xiaoming Gu

Department of Computer Science
University of Rochester
Rochester, NY 14627

{bai,cding,xiaoming}@cs.rochester.edu

Jonathan E. Bard *

NYS Center for Excellence in
Bioinformatics and Life Sciences
SUNY-Buffalo
Buffalo, NY 14203

jbard@buffalo.edu

Abstract

To achieve optimal performance, garbage-collected applications must balance the sizes of their heaps dynamically. Sizing the heap too small can reduce throughput by increasing the number of garbage collections that must be performed. Too large a heap, however, can cause the system to page and drag down the overall throughput. In today's multicore, multiprocessor machines, multiple garbage-collected applications may run simultaneously. As a result, each virtual machine (VM) must adjust its memory demands to reflect not only the behavior of the application it is running, but also the behavior of the peer applications running on the system.

We present a memory management system that enables VMs to react to memory demands dynamically. Our approach allows the applications' heaps to remain small enough to avoid the negative impacts of paging, while still taking advantage of any memory that is available within the system. This memory manager, which we call *Poor Richard's Memory Manager*, focuses on optimizing overall system performance by allowing applications to share data and make system-wide decisions. We describe the design of our memory management system, show how it can be added to existing VMs with little effort, and document that it has almost no impact on performance when memory is plentiful. Using both homogenous and heterogenous Java workloads, we then show that Poor Richard's memory manager improves average performance by up to a factor 5.5 when the system is paging. We further show that this result is not specific to any garbage collection algorithm, but that this improvement is observed for every garbage collector on which we test it. We finally demonstrate the versatility of our memory manager by using it to improve the performance of a conservative whole-heap garbage collector used in executing .Net applications.

Categories and Subject Descriptors D.3.4 [Processors]: Memory management (garbage collection)

General Terms Experimentation, Measurement, Performance

* Work performed at Canisius College

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

ISMM'11, June 4–5, 2011, San Jose, California, USA.

Copyright © 2011 ACM 978-1-4503-0263-0/11/06...\$10.00.

Keywords poor richard's memory manager, garbage collection, multiprogramming, throughput, paging

1. Introduction

Today's developers are increasingly taking advantage of garbage collection (GC) for the many software engineering benefits it provides. Developers can do this using garbage-collected languages (e.g., Haskell, Java, ML), conservative collectors (e.g., the Boehm-Demers-Weiser collector [19]), or scripting languages executing on a garbage-collected virtual machine (VM) (e.g., JRuby or Groovy). While one study found that programs can be executed just as quickly whether they use garbage collection or explicit memory management, this performance comes only when the garbage-collected heap has been sized appropriately [27]. But keeping the heap an appropriate size is extremely difficult: collecting the heap too early increases GC overhead, but allow the heap to grow too large harms performance by decreasing data locality and increasing the number of TLB misses [13].

Even worse than decreasing locality, allowing a heap to grow too large could mean that it no longer fits in RAM and must have pages evicted to disk. Accessing memory that has been saved back to disk can require six orders of magnitude more time than if it were in main memory. Even if the mutator does not touch the evicted pages, it can still hurt performance during garbage collection. During whole-heap collections, the GC examines **all** of the reachable data. Because the mutator can only use reachable objects, the garbage collector's working set is at least as large as the mutator's, if not larger. A GC can therefore cause a massive slowdown due to paging despite the mutator not suffering any page faults. Thus selecting when to collect the heap depends on many factors: the program, the garbage collector, and the resources available.

As a result of the difficulty selecting a proper heap size, many workarounds and optimizations have been proposed. For developers, the simplest approach is to require users select a static heap size that fits into memory. Others have studied using heap and GC statistics or program profiling to determine the optimal times to collect the heap (e.g., [2, 5, 14, 20, 23, 37, 41, 43]). While these approaches can provide significant improvements when memory is plentiful, they often make poor decisions when resource-bound because they ignore paging's costs.

Other approaches have enhanced the OS to enable making *resource-based* garbage collection decisions which also factor in paging costs when selecting a heap size (e.g., [6, 7, 26, 28, 44–46]). While the scope of their changes varies greatly, all these projects require some additional OS and JVM features. The need for these

features limits the ease of adopting them in new or different environments. Another problem is the multiprogrammed environments that are common today. In these situations, the amount of memory available changes dynamically and unpredictably. When these changes are observed by like-minded programs, they would choose to react similarly. Should the programs follow a conservative approach, resources may be left underutilized. More aggressive approaches, however, set a collision course to severe contention and poor performance. Unfortunately, this earlier research assumes an application was running on a dedicated machine and did not evaluate how it would perform when this was not the case.

Contributions: In this paper we address this problem of shared resource utilization by introducing *Poor Richard's Memory Manager*. Poor Richard's handles both the adjustments needed by a single process on a dedicated system and those needed to achieve good performance in the chaotic environments created by executing multiple processes. Poor Richard's provides this improvement by having each process monitor its own performance. Each process also has access to a shared *whiteboard*, in which they can each post information about their own state and read information about others' states. This allows processes to make a joint response to dynamic events and ensure that it can make a system-wide decision about this shared resource.

Following ideals espoused by its namesake, Poor Richard's uses a frugal, lightweight approach to perform its tasks. Applications monitor their state using only information already made available by most operating systems, thus avoiding any changes or additions to the OS. Similarly, Poor Richard's needs only limited interactions with the virtual machine and is entirely independent of the garbage collector being used. The code used to trigger resource-based collections in the conservative whole-heap Boehm-Demers-Weiser collector [19] is the same as the code used to trigger resource-based collection in the generational collectors defined by MMTk. This combination makes it very simple to port and adopt Poor Richard's memory manager in any system.

We present an empirical evaluation of Poor Richard's performance using multiple garbage collectors with both heterogeneous and homogeneous workloads and on two different architectures. We show that it has minimal effect on performance when there is no memory pressure. We further demonstrate that Poor Richard's provides significant performance improvements with every garbage collector we tested when memory pressure increases. Because it limits the effects of paging in an orthogonal manner to how the VM already sizes its heap, Poor Richard's enables systems to use their existing heap sizing algorithms without worrying about how they will perform if the amount of available memory suddenly changes.

The rest of the paper is organized as follows: Section 2 provides an overview of Poor Richard's Memory Manager and how it coordinates a system-wide response to memory pressure. Section 3 describes the implementation of Poor Richard's and its *whiteboard*. The results from our empirical investigation of Poor Richard's are found in Section 4. Section 5 discusses related work, and Section 6 discusses future directions for this research and concludes.

2. Overview of Poor Richard's

Poor Richard's must first be used for it to ever be useful. Adoption of this memory manager is unlikely if it required executing additional processes or adding to or modifying the OS. Even needing substantial changes to the virtual machine can limit the desire to adopt a system. We therefore made the decision that Poor Richard's would make its decisions independent of the virtual machine and that any work which needed to be performed would be done by processes using Poor Richard's only. This simplifies porting it to new environments easy as it relies only on information already available

and data common to all garbage-collected applications and requires minimal changes to the VM.

The goal of Poor Richard's memory manager is to eliminate the need to consider paging when selecting a heap size. This requires allowing processes to utilize as much main memory as they need, but not allow them to use so much that paging limits throughput. This problem is difficult on a dedicated machine since it requires balancing the mutator's and GC's working sets and adapting to changes in mutator behavior. When multiple processes are executing, the problem only gets harder. Any solutions must now account not only for changes in other processes' behavior, but also the possibility of other processes starting and stopping. Just modifying heap sizes may not be enough, since new demands may trigger paging before the next collection yields a chance to shrink the heap size. Poor Richard's solves this dilemma by working orthogonally to heap size selection. Instead it performs periodic checks and, when necessary, triggers an immediate whole-heap "resource-driven" GC. If memory suddenly becomes available, Poor Richard's will not trigger a GC and can continue using the optimal heap size previously computed.

Poor Richard's works orthogonally to systems' existing heap sizing algorithms; the resource-driven collection works like an immediate one-time shrinking of a process's heap. Because this collection is performed as soon as memory pressure is detected, it only needs to maintain the existing level of memory pressure to avoid significant slowdowns. By finding and freeing garbage objects in the heap, the collection ensures future allocations goes to pages that already contain live objects and so avoid the need to page. If the collector also performs compaction or frees pages from the heap or containing metadata, then the collection can not only prevent paging, but even reduce memory pressure. Because the prevention of paging occurs as a result of the garbage collection and does not rely on any specific algorithm, resource-driven collections should improve the performance of any system from those using whole-heap conservative collectors to those using compacting, generational collectors.

By working in multiprogrammed environments, our system must handle an additional concern. As they are running, all processes will see the same stimuli and react identically. If processes are able to react fast enough this may not be a problem, but this could lead to resources being underutilized from overly-conservative decisions or lead to contention as a result of hyper-aggressive choices. An alternative is to allow processes to work collaboratively and develop a system-wide response. As machines feed their increasing numbers of cores by execute more processes simultaneously, we feel coordination is important *if* the costs of coordinating the response can be limited. Poor Richard's therefore includes a mechanism by which processes can coordinate a strategy. To test this idea empirically, we implemented several different coordination strategies with different coordinating costs:

Selfish The simplest coordination strategy is to not coordinate at all. *Selfish* runs of Poor Richard's do not use any coordinating mechanisms and make decisions independently. When running, these processes perform a whole-heap collection whenever they detect memory pressure and do not communicate with other processes.

Communal A second approach to coordinating a response is to have processes share fully in any necessary responsibilities. When using a *Communal* strategy, processes notify all others when they detect memory pressure. When they check for memory pressure, processes also look for these notifications. Whether the process detects memory pressure itself or is notified that it was detected by another, the process performs a whole-heap collection.

If multiple cooperative applications began collecting their heap at the same time, the increased memory demands could itself

trigger further paging and contention accessing the disk. Poor Richard's prevents this by using a flag placed in a shared memory buffer (the "whiteboard") to record whenever a process is performing resource-driven collection. While a process performs a resource-driven collection, others will continue executing normally, including performing demand-driven collections. Once the resource-driven collection completes, the process clears the shared flag and allows another to perform its resource-driven collection. Thus the need to reduce memory pressure is shared while the collections are serialized to prevent clustered collections from overloading the system.

Leadered Our third approach tries reducing the overheads of the Communal strategy while providing a more systemic solution than the Selfish strategy. Instead of all processes collecting their heaps once memory pressure is detected, a process detecting memory pressure signals a single process (the "leader") to perform a whole-heap collection. This ensures only one process collects its heap, thereby reducing memory demands while needing minimal overheads. How the leader is chosen can be tailored to the goals or needs of the machine. So long as some process collects their heap, memory pressure is reduced and the performance is not hurt. In this work, we evaluated selecting the process with the largest heap size as the leader.

3. Poor Richard's Memory Manager

The original Poor Richard espoused the ideals of frugality, efficiency, and unobtrusiveness. We adopted those ideals in creating a lightweight system that can easily be added to virtual machines to preserve throughput even when systems becomes resource-bound. In this section, we discuss the different techniques Poor Richard's Memory Manager employs to meet these ideals. We will initially present Poor Richard's frugality in having applications detect memory pressure only using information the operating system already provides and without knowledge of the VM or GC. We then discuss our implementation and use of the *whiteboard* to allow processes to communicate and cooperate. Finally, we document Poor Richard's interactions with the host VM and how it communicates its decisions.

3.1 Process Self-Monitoring

In Poor Richard's, each process is responsible for monitoring its own state for signs of memory pressure. In particular, processes track how many *major page faults* (evicted pages that have been read back into memory) they trigger and their *resident set sizes (RSS)* (number of pages physically residing in main memory). Poor Richard's relies upon the number of major page faults occurring since the end of the last GC as its primary indicator of memory pressure. Multiprogrammed systems running multithreaded processes can tolerate low levels of major page faults by executing the processes and threads that are not blocked by I/O. Being overly conservative, therefore, increases overheads without improving throughput. Only when the number of major page faults passes a threshold value, for this paper we used 10, will Poor Richard's report it detected increased memory pressure and take action.

As Grzegorzczuk et al. correctly noted [26], however, processes seeing major page faults can tell that the system is resource-bound, but lack knowledge of the source of this contention. We further observe that major page faults can only be detected AFTER memory pressure not only caused pages with usable data on them to be evicted to disk, but that the data they contain was again needed by the mutator or GC. As Poor Richard himself noted, "the early bird catches the worm"; by reacting to memory pressure earlier, systems can avoid making the situation even worse. Towards this end, Poor Richard's also checks for changes in the process's resident set size.

```
extern "C" int checkMemoryPressure() {
    long long recentPFaults;
    long currentRSS;
    long deltaRSS;

    recentPFaults = getPageFaults() - lastPFaults;
    currentRSS = getResidentSetSize();
    deltaRSS = currentRSS - lastRSS;
    lastRSS = currentRSS;
    return ((recentPFaults >= 10) || (deltaRSS < 0));
}
```

Figure 1. Example code by which Poor Richard's memory manager determines if memory pressure warrants further action. While this assumes the VM does not relinquish pages, it could be handled by comparing `deltaRSS` with the number of relinquished pages.

For performance or correctness reasons, most VMs do not willingly relinquish pages; a decrease in the RSS means resources were limited enough to require evicting the process's pages.¹ Without making significant changes to the OS or VM this metric is imperfect: it fails to detect a problem when the process allocates or faults in more pages than are evicted. As we found, however, RSS frugally provides an early notification of memory pressure that would otherwise go unnoticed.

By focusing on being frugal, adopting Poor Richard's memory manager is made easier. Figure 1 shows the code it executes to check whether increased memory pressure requires further action. The only environment-specific portion of this code are the methods finding the process's current count of major page faults and resident set size. In Linux, processes can find these values among the performance statistics available in `/proc/self/stat` [4].² As part of the `/proc` directory, this pseudo-file can be opened and read like a normal file, but is really an interface for processes to access data from the kernel [42]. Thus Poor Richard's need not rely on any OS-specific features or modifications and can be easily ported.

3.2 Whiteboard Communication

In the multiprogrammed environments that are frequently used, each process's decisions about shared resources impact the other processes being executed. Consider, for example, the effect of a process performing a whole-heap collection. Whereas the mutator's working set may include only a subset of reachable data, the GC must examine all reachable objects. As a result, garbage collection will temporarily increase the memory needs of that process. Should multiple processes using resource-based collection detect increased memory pressure at the same time, their combined increased memory demands could result in severe paging and contention accessing the disk.

Were processes able to communicate, problems such as these could be avoided. As we expect memory pressure and paging to not be the common case, it is vital that collaborations be as efficient as possible. As in Wegiel and Krintz [40], Poor Richard's uses a shared memory buffer to facilitate efficient interprocess communication. Into this buffer, Poor Richard's allocates a common "whiteboard". Processes can then use this whiteboard to share important information and coordinate their actions. When they begin executing, processes can load the whiteboard into their address space and register themselves as a participant. As part of their registration

¹This would also work with VMs that voluntarily relinquish memories (e.g., HotSpot with ergonomics [1]), by allowing the VM to specify the number of pages it gave up and returning if the RSS decreases by more than this expected amount.

²While `/proc/*/stat` exists only in Linux, these values are obtainable in Solaris, AIX, and other Unix systems using the `getrusage` function [3].

with the whiteboard, each process takes a space of its own on the whiteboard. At the end of each whole-heap collection, the process updates this space to specify its heap size, resident set size, and number of page faults. Each private space in the whiteboard also contains a flag that other processes can set to get that process's attention. The whiteboard also contains a shared data area in which are stored data needed for bookkeeping and to prevent data races. This shared area also contains a flag that processes can set while they perform a whole-heap collection to alert others that they will be increasing memory pressure on the system temporarily. Prior to termination, processes remove themselves from the whiteboard and make their private space available for use by another process. The whiteboard's memory demands are very limited needing only 848 bytes to hold the common values and 32 bytes for each private space.

Use of this whiteboard is therefore very efficient. Absent memory pressure, processes need only access shared areas only twice – when they register and unregister with the system – and only update their private space at the completion of each whole-heap GC. Using Poor Richard's whiteboard, processes can notify others when they detect memory pressure and allow easy implementation of any of the coordination strategies presented in Section 2. No matter the coordination strategy, Poor Richard's whiteboard enables this communication to occur efficiently and with a minimum of overhead.

3.3 Interaction with the VM

All of Poor Richard's frugality would be for naught were existing VMs required to make substantial changes to use it. The final ideal of our design was unobtrusiveness: that Poor Richard's memory manager require minimal changes and few interactions with the host VM. By doing this, we maintain all of a VM's existing optimizations and tuning to preserve their existing good performance when memory pressure is low.

These minimal changes are possible because most of the memory manager resides in a separate, fully independent, library of "C" code that gets compiled into the VM when the VM is built. Within this library resides all of its whiteboard functionality as well as the code with which a process monitors itself for signs of memory pressure. This library also contains the code which analyzes all of this data, coordinates actions with other processes, and determines the appropriate action for the VM.

To ensure Poor Richard's remain unobtrusive, code in the garbage collector is responsible for initiating all communication. The first of the calls from the GC to Poor Richard's is added to the end of the GC code responsible for reclaiming space following a whole-heap collection. When this call is made, Poor Richard's updates its baseline values of both the number of major page faults seen by this process and the process's resident set size. These are the values that the system uses to determine when a process is seeing sufficient memory pressure to warrant a response (the variables `lastPFaults` and `lastRSS` in Figure 1). During this call, Poor Richard's also updates the data stored in the process's private area of the whiteboard (i.e., the number of major page faults seen, resident set size, and the size of the heap after heap memory was reclaimed). As the purpose of this call is purely informational, no data is returned and, once complete, the GC continues as normal.

The second of the calls from the GC is where Poor Richard's can modify the behavior of the VM. This call allows Poor Richard's to check for rising memory pressure and require the GC to act when it is necessary. An example of the code executed by Poor Richard's during this call is shown in Figure 2. How the calls interacting with the whiteboard are defined depends on coordination strategy being used. For Selfish runs of Poor Richard's, these functions did not use the whiteboard, but instead checked a global variable. For other runs, they would use the whiteboard. This makes this code

```
extern "C" int consultPoorRichards() {
    int forceGC = checkWBFlags();
    if (!forceGC) {
        int memoryPressure = checkMemoryPressure();
        if (memoryPressure) {
            setWBFlags();
            forceGC = checkWBFlags();
        }
    }
    return forceGC;
}
```

Figure 2. Example of Poor Richard's code that determines if a process should perform a whole-heap collection. This initially checks if any external process has flagged this process to perform a resource-driven collection. If not, it checks for memory pressure. If memory pressure is detected, the current coordination strategy selects the process(es) chosen to perform a resource-driven collection. Finally, it checks again to see if the current process was among those selected to perform a resource-driven collection.

```
public final static boolean USING_PRMM = true;
public static int waitConsult = 100;
public static int slowPathsWait = 0;
public final boolean gcCheck() {
    int nurseryPages = nurserySpace.reservedPages();

    if (nurseryPages >= Options.nurserySize.getMaxNursery() ||
        nurseryPages >= getMatureSpacePagesAvail()) {
        return true;
    } else if (USING_PRMM && ++slowPathsWait >= waitConsult) {
        boolean forceGC = consultPoorRichards();
        waitConsult = computeNextDelay();
        slowPathsWait = 0;
        forceFullHeapCollection = forceGC;
        return forceGC;
    }
    return false;
}
```

Figure 3. Example of a collector's "slow-path" allocation code modified to also check with Poor Richard's memory manager. With this addition, the routines which already trigger demand-driven collections will also trigger any resource-driven collections.

very easy to modify and update with new coordination strategies. `consultPoorRichards` begins by checking the private area of the whiteboard to see if another process detected memory pressure and, following the active coordination strategy, determined that the current process must reduce its working set. When selected to perform a whole-heap collection, Poor Richard's returns immediately and notifies the GC of this need. If it is not notified that it must collect its heap, we perform the self-monitoring process from Figure 1. If this monitoring does not detect any memory pressure (i.e., `checkMemoryPressure()` returns false), Poor Richard's is done and allows the GC to continue as normal. When memory pressure is detected, Poor Richard's applies the active coordination strategy, notifies any processes which will need to perform a whole-heap GC, and returns if it was the selected process. How the calls interacting with the whiteboard are defined depends on coordination strategy being used.

The call in which the VM consults Poor Richard's is appended to the GC's existing "slow-path" allocation routine. Within this slow-path routine the GC already checks for demand-driven collections and so is a natural place to trigger resource-driven collections. An example of this modified slow-path routine is in Figure 3. The call to Poor Richard's is additive only and will not interfere or override any demand-driven collection decisions. When resources are plentiful, Poor Richard's never signals for a collection and so

the system executes as normal. Even when resources are limited, Poor Richard's will not reject any collections or modify the preferred heap size. Instead Poor Richard's directs a process to collect its heap, thereby shrinking the working set and reducing memory pressure. If more resources become available immediately following a resource-driven collection, Poor Richard's memory manager would revert to not signalling collections and the process would continue executing as before.

Each time the VM calls Poor Richard's to check if a demand-driven collection is necessary there will need to be multiple function calls, an interaction with kernel data, and access to several buffers used only for this process. The overhead of this process is cheap relative to the cost of a hard drive access, but calling this too frequently would harm performance when memory is plentiful. Thus the VM will not perform on each slow-path allocation, but instead makes this call only periodically. This rate is determined using an additive increase/multiplicative decrease algorithm. Using this heuristic, whenever a process detects memory pressure in its self-monitoring it immediately reduces the number of slow-path allocator calls between consultations (i.e., `waitConsult` in Figure 3) by an order of magnitude; when no memory pressure is detected the rate is decreased by 1. This approach allows Poor Richard's to respond quickly when a response appears likely, but limit overheads when memory is plentiful.

Just as being frugal makes it easy to port our memory manager to many different environments, being unobtrusive simplifies the task of adding the system to an existing VM. The needed changes to the VM require finding where a GC releases memory at the end of a whole-heap collection and the allocation's slow path. The code within Poor Richard's cannot depend on any implementation details of the VM in which it runs and will not require any existing VM code be modified. As with the other ideals espoused by the its namesake, remaining unobtrusive keeps the implementation of Poor Richard's lightweight and ready to be included within a VM with the addition of only a few lines of code.

4. Results

We now present the results of our empirical analysis of Poor Richard's Memory Manager. We will describe our experimental methodology including the environments and garbage collectors with which we perform the majority of our experiments. We then show how Poor Richard's Memory Manager improves paging performance on multiple garbage collectors and on multiple architectures for both heterogenous and homogenous workloads. Finally, we discuss porting Poor Richard's to the Mono runtime and show how it also improves the throughput of .Net benchmarks.

4.1 Implementation

For all of our Java experiments, we used the Jikes RVM/MMTk, version 3.1.1. Nearly all of our implementation of Poor Richard's was written as a separate library of C code, however. This includes all of the code detecting memory pressure, accessing the whiteboard, and determining whether a process should perform a collection or not. Calls to Poor Richard's required using `SysCall` routines to support the Java-to-C transition. Only two other additions were made to the Jikes RVM/MMTk. First, we modified the slow-path allocation routines to consult with Poor Richard's and allow it to trigger resource-driven whole-heap collections. Second, we added support within the garbage collectors to allow Poor Richard's to update the process's private whiteboard space with the new heap size following each whole-heap collections. Because of the simplicity of these interactions, and despite needing each call to go from Java to C and back, these changes required under 200 LOC.

To see how well our approach would work with other systems and garbage-collection idioms, we also tested Poor Richard's using

the Mono VM, an open source virtual machine which executes applications written for .Net [32]. Because Mono is already written in C, we were able to call directly into Poor Richard's functions without any overhead. As a result, we needed under **10** lines of code to enable our system in Mono and did not require any modifications to our memory manager.

4.2 Methodology

To determine how well Poor Richard's will work in multiprogramming environments, we performed our experiments on two separate machines. The first machine contains two processors each of which is a single-core, hyperthreaded 2.8GHz Intel Xeon processor with 1MB of L2 cache. For all of our unconstrained memory experiments, we used the machine's full 4GB of RAM. To create memory pressure, we used the `grub` loader to limit the system to only recognize 256MB of physical memory. Our second machine contained a four-core, 2.6GHz Intel Core2 processor with 4MB of L2 cache. To create memory pressure on this machine, we again used the `grub` loader to limit the system to 512MB of physical memory. Both system use a vanilla Linux kernel version 2.6.28. During these experiments, each machine was placed in single-user mode with all but the necessary processes stopped and the network disabled. Experiments on the first machine ran two applications concurrently while experiments on the second machine ran four applications concurrently. We ran as many application as there were machine cores, but allowed the applications to execute normally and did not limit or bind them to a core. For each experiment, we record the time required until all of the processes completed.³

For these experiments, we used benchmarks drawn from two sources. The first benchmark was `pseudoJBB`, a fixed workload variant of `SPECjbb` [38]. In addition, we used four benchmarks from the 2006-10-MR2 release of the DaCapo benchmark suite: `bloat`, `fop`, `pmd`, and `xalan` [15].

To limit variations between runs, these experiments used a pseudo-adaptive compilation methodology [29, 35]. Under this approach, we initially timed five separate runs of each benchmark when executing with the adaptive compiler and recorded the final optimization decisions made in each run. Using the decisions from the fastest of these five runs, all of our experiments duplicate the decisions of an adaptive compiler strategy but in an entirely repeatable manner. Compilation costs can still bias results of experiments [24]. As a result, most experiments follow a second run methodology [9]. We observed, however, that the first ("*compilation*") pass can trigger significant paging which influences the results of further passes. Since we could control or eliminate the effects of this compilation pass, we chose instead to loop the benchmark so that it ran at least five times to minimize biases introduced by the compilation costs.

Using these benchmarks we ran two sets of experiments. The first set of experiments examined system throughput when processes are executing homogeneous workloads. For these workloads, we timed how long was needed to completely execute two or four parallel instances of the benchmark. For our second set of experiments, we analyzed system performance on heterogeneous workloads using each possible pairing of DaCapo benchmarks. When running four applications, we had two processes execute one of the benchmarks and another two processes running the other. While these benchmarks would normally complete in a different amount of time, we selected the number of times each benchmark was looped so that they would complete within approximately one

³We also examined the results using the average completion time of each process. This did not change our results or conclusions and so is not presented here.

Benchmark	Bytes Alloc.	Min. Heap	# Passes	Description
pseudoJBB	0.92G	42M	1	Java server benchmark
bloat	684M	22M	5	Java bytecode optimizer
fop	66M	24M	23	Translate XSL-FO to PDF
pmd	322M	20M	7	Java program analysis
xalan	77M	99M	8	XSLT transformer

Table 1. Key Metrics for Single Pass of Each Benchmark

second of each other when run with no memory pressure. Table 1 provides important metrics for each of these benchmarks.

We ran five trials per data point and report the mean of each set of trials. Our experiments found high variances for the results of the default system. These variances decreased for Poor Richard’s runs using the Collaborative strategy. For Poor Richard’s runs using either the Selfish or Leadered strategy, the variances were minimal. In our results, the variance appears correlated to the amount of paging that occurs. This provides further evidence of how well Poor Richard’s handles memory pressure.

4.3 Performance without Memory Pressure

While the rate with which the system consults with Poor Richard’s does grow slowly, it will not go away. To maintain good performance, it was important that the processing performed by Poor Richard’s be as frugal as possible. When we tested the systems using the full 4GB of RAM, we saw that this is the case. Using GenImmix and averaged across all our experiments, there is no meaningful difference between the time executing any of Poor Richard’s coordinating strategies and the base Jikes RVM runs – the differences are too small to be distinguishable from the background noise. While the differences are slightly greater for GenMS, the base system averages only up to 1.6% faster than using Poor Richard’s with the Leadered coordinating strategy. As Poor Richard’s using the Communal strategy is nominally faster, on average, than the base Jikes RVM at the same heap size, it suggests that this result is probably within experimental error as well.

4.4 Performance with Memory Pressure

We now consider the performance of Poor Richard’s Memory Manager when under memory pressure. We first examine whether Poor Richard’s improves the performance of the two best performing collectors included within Jikes RVM/MMTk: generational collectors using either a mark-sweep policy (“GenMS”) or the Immix algorithm [16] to collect its mature space (“GenImmix”). Figure 4 uses the dual-processor machine to compare the performance of Jikes RVM with runs with Poor Richard’s using either the Selfish strategy, the Communal strategy, or a Leadered strategy. To insure that improvements are due to Poor Richard’s memory manager only, these graphs (and all results in this paper) are for runs using fixed heap sizes.⁴ In these graphs, the x -axis shows the fixed heap size used for the run and the y -axis shows the execution times relative to the performance of Poor Richard’s memory manager using the Leadered strategy.

As soon as paging begins, the benefits of our approach become immediately apparent. The performance of runs using either the basic GenMS or GenImmix GCs quickly degrade as a result of paging. As Figure 4(a) shows, this degradation can lead to factor of 10 or more slowdowns. The results from the Poor Richard’s runs, however, show that all of its strategies can reduce this slowdown

⁴ We repeated these experiments with runs using the default heap sizing algorithm and a fixed upper-bound on the heap size. These showed similar relative results and support these conclusions.

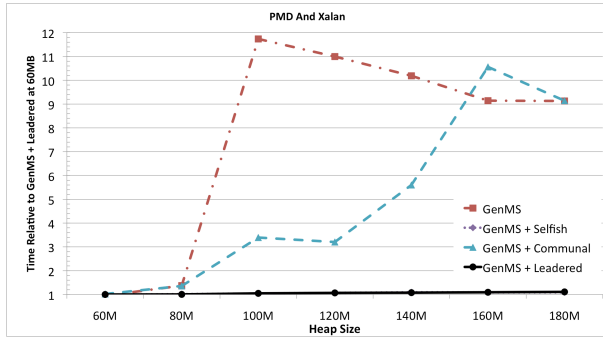
considerably. The Communal strategy is consistently the worst performing strategy for Poor Richard’s. This poor performance comes about for multiple reasons. Forcing all processes to collect their heap in response to memory pressure, even with those collections being serialized, works only when executing homogeneous workloads (Figures 4(c) and 4(d)) or when both processes have heap sizes that greatly exceed the size of the live data in the heap (Figure 4(b)). When processes have very different workloads, few of the GCs triggered by the Communal strategy are needed and the short-term increase in process’ working set sizes leads to greater amounts of paging. This results in performance that can be worse than the default approach (Figure 4(a)). As our results from the Leadered strategy show, only one process needs to GC to alleviate memory pressure. Second and subsequent resource-driven collections merely add GC overhead and are not needed to prevent paging.

The runs using Poor Richard’s with either the Selfish or Leadered strategies, in contrast, provide very consistent results across all benchmarks. Because these results are for runs using only two processes, it is unlikely that both processes would perform simultaneous resource-driven GCs; one would expect the performance of the these strategies to be very similar on this architecture. As can be seen in Figure 5, this is the case. While the Leadered strategy was slightly better, the differences seen between these two strategies were within the variances we measured. A more interesting note is that the performance of both these collectors was largely independent of the heap size specified. At the largest heap sizes, the Leadered strategy averaged throughput a factor of 1.07 slower than at the smallest heap size using GenMS and a factor of 1.04 slower using GenImmix. For the Selfish strategy, the slowdowns were roughly similar: a factor of 1.06 for runs with GenMS and a factor of 1.05 slower for runs using GenImmix. This suggests that either of these approaches work well towards eliminating paging concerns when selecting a heap size on a dual-core machine.

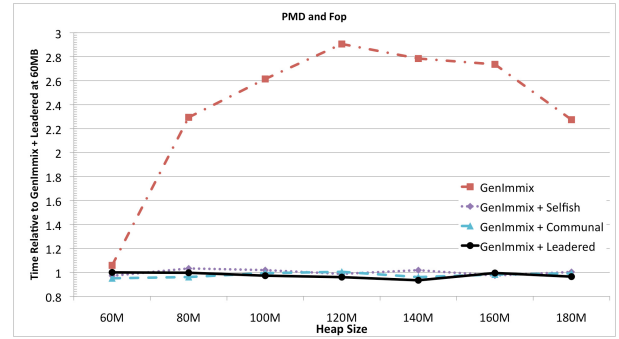
In Figure 6 one can see the results from 4 simultaneous runs of pseudoJBB using the GenImmix collector. These demonstrate that the problems that arise with paging may only get worse as we use machines with ever increasing numbers of cores. As when two processes were simultaneously executed, paging causes significant slowdowns to the default Jikes RVM system. As one might expect, the overhead imposed by Communal strategy becomes increasingly more expensive as we add processes. While this system-wide solution to paging may seem attractive, its strength of involving all processes in the solution also means it is unable to respond quickly enough to avoid paging or even alleviate paging that might occur. Similarly, these results begin to show the weakness of the Selfish approach. With more processes executing, we begin to see instances where two processes perform resource-driven GCs at the same time. As we predicted, this increases the memory pressure on the system and actually causes more paging, not less. Because this event does not always happen, the results for the Selfish strategy become more erratic than before. When everything works the Selfish strategy can still match the Leadered strategy, but only if everything falls into place at the correct times. Figure 7 shows that these findings continue to hold across the board. By collaborating on a system-wide response, but without significant overheads, the Leadered strategy continues performing well at all heap sizes.

4.5 Performance in Mono

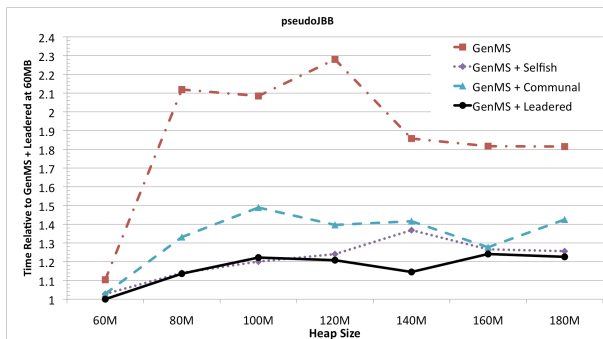
As a last experiment to see how adaptable our approach really was, we also ported Poor Richard’s to the Mono [32] VM which executes .Net applications. This port was interesting because Mono uses a conservative, whole-heap collector based upon the BDW collector and is very different from the GCs on which we previously tested Poor Richard’s. The differences were not hard to overcome,



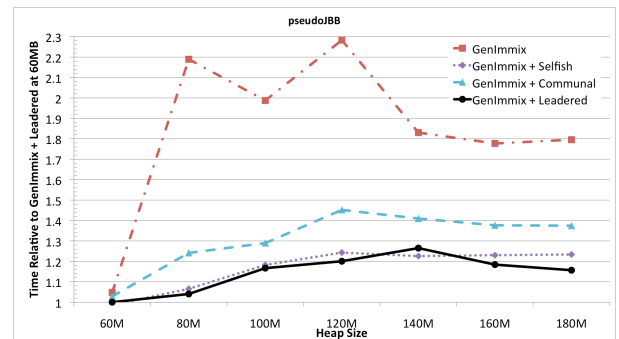
(a) Executing Xalan and PMD for GenMS and GenMS with Poor Richard’s using 3 different strategies. While all runs using Poor Richard’s memory manager outperforms the default system, runs using the Selfish or Leadered strategies do substantially better.



(b) Relative performance executing Fop and PMD for GenImmix with Poor Richard’s using 3 different strategies. The performance of Poor Richard’s memory manager does not depend on the GC being executed. When heap sizes get large enough that the number of GCs decreases substantially, the base Jikes RVM does not suffer as much from paging.



(c) Relative throughput executing pseudoJBB for GenMS and GenMS with Poor Richard’s using 3 different strategies. While the Communal strategy does better with these homogeneous workloads, the Selfish and Leadered strategies continue to outperform the other approaches.



(d) Relative throughput executing pseudoJBB for GenImmix and GenImmix with Poor Richard’s using 3 different strategies. Even with only 2 processes running in these experiments, the slight collaboration provided by the Leadered strategy can improve overall throughput.

Figure 4. Graphs showing that Poor Richard’s memory manager improves the paging performance of both the GenMS and GenImmix collectors in Jikes RVM on a dual processor machine for both homogeneous and heterogeneous workloads. Because paging’s impact affects all processes, a system-wide response can improve performance even when only two processes are executing.

however. Because of the simplicity of our interface, this port was very simple to make. Much more difficult was finding benchmarks we could run that were capable of generating a heap that could trigger paging. In the end, we relied on a port of the GCOld synthetic benchmark. We used this with several different ratios of short-to-long lived objects to test our system. Figure 8 shows that Poor Richard’s memory manager was even able to improve the throughput of 2 simultaneous executions of this system, offering a speedup between 1.5 and 1.73.

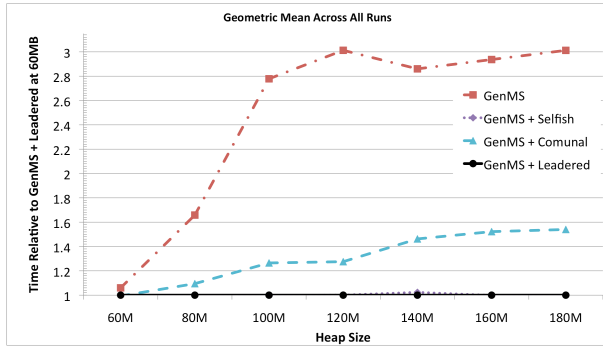
5. Related work

The idea of using a lightweight approach to manage shared resources was first investigated by Zhang et al. [47]. This work investigated ways of handling threads within virtual machines executing in multiprogrammed environments. Each “friendly” virtual machine estimates system load using information already made available by the OS. Each process works selfishly, suspending threads upon determining the system is overloaded and resuming threads when it finds the system can handle an increased load. The earlier work focuses on optimizing system performance for embarrassingly parallel processes in which each thread can be treated as equal and independent. Our work investigates a more complex and dynamic environment with a far greater penalty for making the

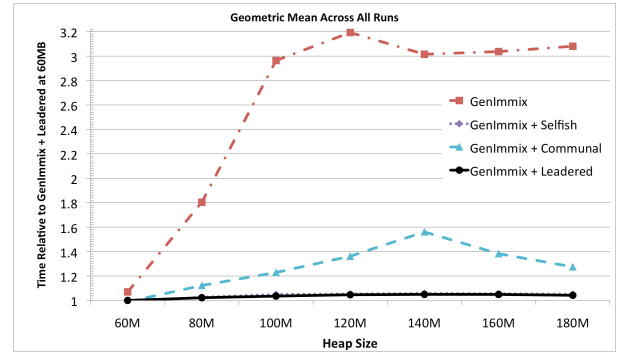
wrong decision. While both works are motivated similarly, the research differs greatly in their environments, goals, and the means of achieving these goals.

Many early garbage collection algorithms included features designed to reduce heap sizes or the effects of paging. One common solution is to use heap compaction algorithms, which reduce the size of a program’s working set and therefore the need for a system to page [10, 11, 17, 18, 22, 25]. As the size of available memory increased, algorithms were proposed to divide the heap into spaces or generations which could be collected individually [8, 12, 31, 33, 34, 39]. Our work, like these algorithms, tries to control the size of the heap to limit applications’ and garbage collectors’ working sets and thus reduce the need for pages to be evicted and the effects of unnecessary paging. Our work differs, however, in that it includes system-wide communication about memory pressure and is orthogonal to the specific GC algorithms being used.

Several recent approaches explored resource-based memory management and used both the operating system and virtual machine to improve paging performance. Yang et al. modified the operating system to develop approximate reuse distance histograms with which they could estimate the current available memory size. Their CRAMM system then developed collector models which enabled the JVM to select a heap size that would utilize available



(a) Relative throughput across all runs for GenMS and GenMS with Poor Richard’s using 3 different strategies. These results show that the Selfish and Leadedered strategies perform well no matter the heap size.



(b) Relative throughput across all runs for GenImmix and GenImmix with Poor Richard’s using 3 different strategies. The Leadedered approach continues to provide good performance that is largely independent of the heap size specified.

Figure 5. Graphs showing that Poor Richard’s memory manager improves the paging performance of both the GenMS and GenImmix collectors in Jikes RVM on a dual processor machine for both homogeneous and heterogeneous workloads. Because paging’s impact affects all processes, a system-wide response can improve performance even when only two processes are executing.

physical memory fully [44, 45]. Hertz et al. developed the Book-marking Collector (BC), a paging-aware garbage collector that worked in conjunction with a modified virtual memory manager. When used together it was able to eliminate paging costs in large heaps and greatly reduce paging costs in all situations [28]. Unlike CRAMM and BC, Poor Richard’s needs no OS changes, interacts minimally with the host VM, and is independent of the GC algorithm being used.

Other studies of resource-based memory management considered ways to size the heap in response to memory pressure. Alonso and Appel presented a collector that followed each collection by consulting a process (the “advisor”). Their system could reduce the heap size to a level based on the amount of available memory [6]. More recently, Grzegorzczak et al. developed a tiny addition to Linux with which they could determine the number of page allocation stalls. Using a count of these stalls, which they showed is a good indicator of memory pressure [26], their Isla Vista system could guide a VM’s heap sizing policy appropriately. Unlike these past works our approach neither modifies applications’ heap sizes nor requires any additions to the OS, but instead relies on frequent polling of already available information. Our work also differs in that it explicitly considers memory sharing by multiple JVMs and enables decision making based upon system-wide information.

Researchers have also investigated using program analysis to reduce the effects of paging on garbage-collected applications. Andreasson et al. used reinforcement learning to improve GC decisions through thousands of iterations. They assigned fixed cost for GC and paging and predicted the running time as a function of these and other parameters [7]. The average performance improvement for SPECjbb2K running on JRockit was 2% with the learning overhead and 6% otherwise. Instead of using a fixed cost and memory size, other recent work [46] adaptively monitored the number of page faults and adjusted the heap size of a program in an exclusive environment. Both of these methods relied upon manual analysis of the program. Unlike those approaches, our work is fully automated and can be applied to general programs. More importantly, Poor Richard’s works in shared environments.

Many other adaptive schemes have been used for garbage collection. Several studies examined adaptation based on the program demand. Buytaert et al. use offline profiling to determine the amount of reachable data as the program runs and generate a list-

ing of program points when collecting the heap will be most favorable. At runtime, they then can collect the heap when the ratio of reachable to unreachable data is most effective [20]. Similar work by Ding et al. used a Lisp interpreter to show that limiting collections to occur only at phase boundaries reduced GC overhead and improved data locality [23]. By using allocation pauses to dynamically detect phase boundaries and limiting collections to these phase boundaries, Xian et al. improved throughput by up to 14% [43]. Soman et al. used profiling, user annotation, and a modified JVM so a program may select which garbage collector to use at the program loading time [37]. Wegiel and Krintz developed the Yield Predictor which estimated the percentage of the heap a whole-heap garbage collection would reclaim and so could be used to skip performing unproductive collections and just grow the heap initially [41]. MMTk [14] can adjust its heap size by analyzing heap and garbage collection statistics and applying a set of predetermined ratios. Similarly, HotSpot [2] and Oracle JRockit [5] can adjust their heap sizes to target a specific pause time, throughput, or heap size. Our work is orthogonal and complementary because it reacts to the changing resource in the system rather than predicting the demand of applications.

While heap management adds several new wrinkles, there are many prior works creating virtual memory managers which adapt to program behavior to reduce paging. Smaragdakis et al. developed early eviction LRU (EELRU), which made use of recency information to improve eviction decisions [36]. Last reuse distance, another recency metric, was used by Jiang and Zhang to avert thrashing [30], by Chen et al. to improve Linux VM [21], and by Zhou et al. to improve multi-programming [48]. All of these techniques try to best allocate physical memory for a fixed subset of the working set, but are of limited benefit when the total working set fits in available memory or when the available memory is too small for the subset. Resource-based memory management, on the other hand, only triggers collections upon detecting significant memory pressure. Because it does nothing in the absence of paging, the heap will grow to take full advantage of physical memory when it becomes available. Our system is therefore able to trigger more frequent collections (avoiding paging costs) dynamically without sacrificing the larger heap sizes (increased memory usage) that would otherwise provide better performance.

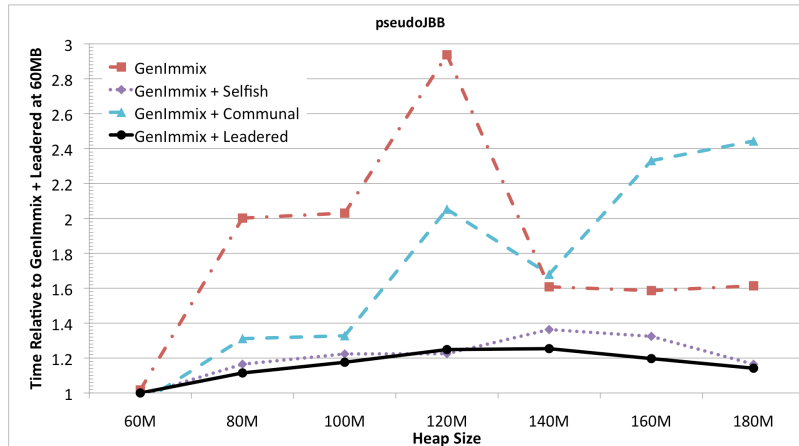


Figure 6. Results when executing 4 processes simultaneously. Only the Leadered strategy’s triggering of a single resource-driven collection in response to memory pressure continues to avoid significant paging.

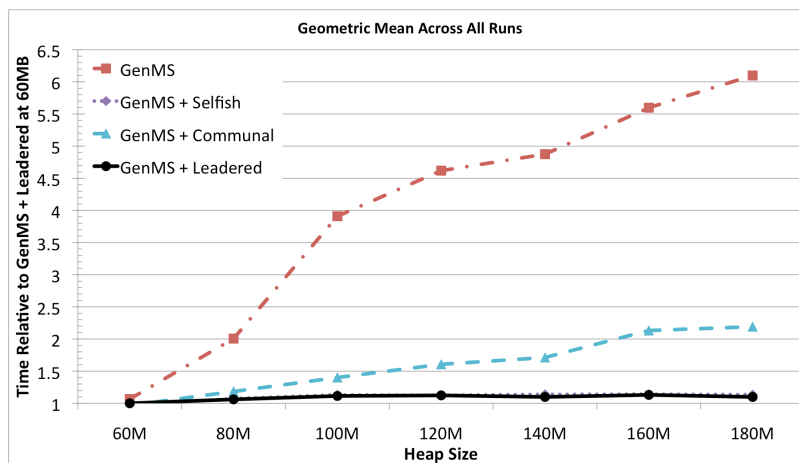


Figure 7. Average across all our runs when executing 4 process simultaneously. Poor Richard’s continues to scale to this environment and the Leadered strategy’s minimal collaboration increasingly outperforms all other approaches.

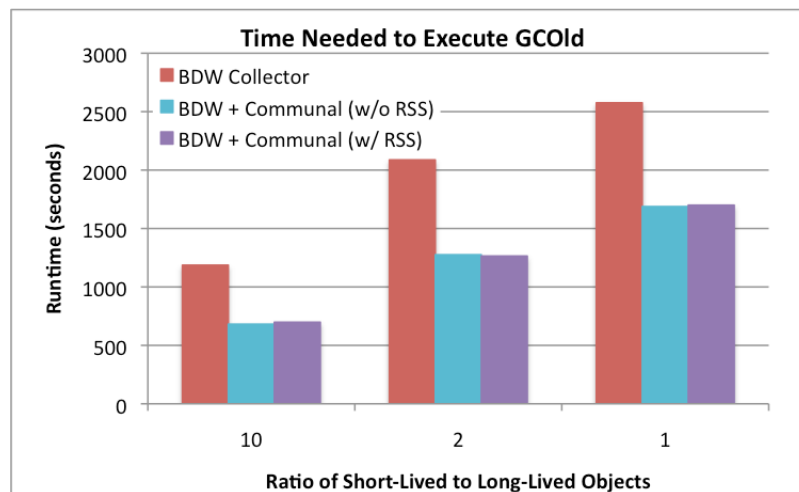


Figure 8. Graphs showing Poor Richard’s memory manager improves the paging performance of the Mono runtime system. Unlike Jikes RVM, Mono uses a conservative, whole-heap collector.

6. Conclusions and Future Work

Garbage-collected applications need their heaps sized to be large enough to provide optimal performance. Sizing the heap too large, however, can lead the system to page and throughput to plummet. When executed on the multi-processor, multi-core machines that are increasingly common, this requires applications adjust their heaps dynamically and in reaction to changes brought on by other processes. Should other processes react to changes in available memory similarly, the result could lead to either increased contention as they all grab for the same resources or decreased utility as they all react too conservatively.

Poor Richard's memory manager avoids this problem by allowing each process to detect and react to increased memory pressure quickly and avoid paging. It does this in a simple, lightweight manner that requires few changes to existing systems, allowing them to keep their existing good performance when not paging, and improves the paging performance of all collectors we tried. By enabling processes to collaborate on system-wide solutions, Poor Richard's scales to multiprogramming environments easily and improves the average throughput across a range of benchmarks by up to a factor of 5.5.

As part of our future work, we plan to port Poor Richard's to additional VMs and investigate alternative methods of selecting a leader in Poor Richard's Leadered strategy. In particular, we plan on investigating approaches that would allow systems to tailor how resources are allocated to meet their particular needs. By providing a mechanism which could be used to ensure certain processes meet guaranteed performance goals, optimize throughput, or allow creation of multiple levels of priority, Poor Richard's could help ensure that the software engineering benefits of garbage collection can continue.

Acknowledgments

We are grateful to IBM Research for making the Jikes RVM system available under open source terms and for the developers of the MMTk memory management toolkit. We would also like to thank Margaret Foster and the anonymous reviewers for their excellent suggestions on this paper.

This work was supported by the Canisius Ensuring Excellence Program and the National Science Foundation under awards CSR-0834566 and CSR-0834323. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

- [1] Bug ID: 4694058 Allow JRE to return unused memory to the system heap. Available at http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4694058.
- [2] Garbage collector ergonomics. Available at <http://download.oracle.com/javase/6/docs/technotes/guides/vm/gc-ergonomics.html>.
- [3] getrusage(3c) - solaris man page. Available at <http://download.oracle.com/docs/cd/E19963-01/821-1465/getrusage-3c/index.html>.
- [4] proc(5) - process info pseudo-filesystem - Linux man page. Available at <http://linux.die.net/man/5/proc>.
- [5] Tuning the memory management system. Available at http://download.oracle.com/docs/cd/E15289_01/doc.40/e15060/memman.htm#i1092162.
- [6] R. Alonso and A. W. Appel. An advisor for flexible working sets. In *Proceedings of the 1990 Joint International Conference on Measurement and Modeling of Computer Systems*, pages 153–162, Boulder, CO, May 1990.
- [7] E. Andreasson, F. Hoffmann, and O. Lindholm. To collect or not to collect? Machine learning for memory management. In *JVM '02: Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 27–39, August 2002.
- [8] A. W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, 1989.
- [9] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Thirtieth Annual ACM Symposium on Principles of Programming Languages*, volume 38(1), pages 285–298, New Orleans, LA, Jan. 2003.
- [10] H. D. Baecker. Garbage collection for virtual memory computer systems. *Communications of the ACM*, 15(11):981–986, Nov. 1972.
- [11] H. G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.
- [12] P. B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, MIT Laboratory for Computer Science, Cambridge, MA, May 1977.
- [13] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and reality: The performance impact of garbage collection. In *Proceedings of the 2004 Joint International Conference on Measurement and Modeling of Computer Systems*, volume 32(1), pages 25–36, June 2004.
- [14] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? high performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, Edinburgh, Scotland, May 2004.
- [15] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, volume 41(10), pages 169–190, Oct. 2006.
- [16] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, volume 43(6), pages 22–32, Tucson, AZ, 2008.
- [17] D. G. Bobrow and D. L. Murphy. Structure of a LISP system using two-level storage. *Communications of the ACM*, 10(3):155–159, Mar. 1967.
- [18] D. G. Bobrow and D. L. Murphy. A note on the efficiency of a LISP computation in a paged machine. *Communications of the ACM*, 11(8):558–560, Aug. 1968.
- [19] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, Sept. 1988.
- [20] D. Buytaert, K. Venstermans, L. Eeckhout, and K. D. Bosschere. GCH: Hints for triggering garbage collections. In *Transactions on High-Performance Embedded Architectures and Compilers I*, pages 74–94. 2007.
- [21] F. Chen, S. Jiang, and X. Zhang. CLOCK-Pro: an effective improvement of the CLOCK replacement. In *Proceedings of USENIX Annual Technical Conference*, 2005.
- [22] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, Nov. 1970.
- [23] C. Ding, C. Zhang, X. Shen, and M. Ogihara. Gated memory control for memory monitoring, leak detection and garbage collection. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance*, pages 62–67, Chicago, IL, June 2005.
- [24] L. Eeckhout, A. Georges, and K. D. Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications*, volume 38(11),

- pages 169–186, Anaheim, CA, Oct. 2003.
- [25] R. R. Fenichel and J. C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, Nov. 1969.
- [26] C. Grzegorzcyk, S. Soman, C. Krintz, and R. Wolski. Isla vista heap sizing: Using feedback to avoid paging. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 325–340, 2007.
- [27] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, volume 40(7), pages 143–153, Chicago, IL, June 2005.
- [28] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 143–153, Chicago, IL, June 2005.
- [29] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications*, volume 39(11), pages 69–80, Vancouver, BC, Canada, Oct. 2004.
- [30] S. Jiang and X. Zhang. TPF: a dynamic system thrashing protection facility. *Software Practice and Experience*, 32(3), 2002.
- [31] H. Lieberman and C. E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.
- [32] Mono Project. Mono. Available at <http://www.mono-project.com/>.
- [33] D. A. Moon. Garbage collection in a large LISP system. In *Conference Record of the 1994 ACM Symposium on Lisp and Functional Programming*, pages 235–245, Austin, TX, Aug. 1994.
- [34] J. H. Reppy. A high-performance garbage collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, Dec. 1993.
- [35] N. Sachindran and J. E. B. Moss. Mark-Copy: Fast copying GC with less space overhead. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications*, volume 38(11), pages 326–343, Anaheim, CA, Oct. 2003.
- [36] Y. Smaragdakis, S. Kaplan, and P. Wilson. The EELRU adaptive replacement algorithm. *Perform. Eval.*, 53(2):93–123, 2003.
- [37] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *Proceedings of the International Symposium on Memory Management*, pages 49–60, Vancouver, BC, Canada, June 2004.
- [38] Standard Performance Evaluation Corporation. Specjbb2000. Available at <http://www.spec.org/jbb2000/docs/userguide.html>.
- [39] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, volume 19(9), pages 157–167, Apr. 1984.
- [40] M. Wegiel and C. Krintz. XMem: type-safe, transparent, shared memory for cross-runtime communication and coordination. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 43(6), pages 327–338, Tucson, AZ, June 2008.
- [41] M. Wegiel and C. Krintz. Dynamic prediction of collection yield for managed runtimes. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, volume 44(3), pages 289–300, Washington, DC, Mar. 2009.
- [42] Wikipedia. procs — Wikipedia, the free encyclopedia, 2011. [Online; accessed 1-Feb-2011].
- [43] F. Xian, W. Seisa-an, and H. Jiang. MicroPhase: An approach to proactively invoking garbage collection for improved performance. In *Proceedings of the 22nd ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications*, volume 42(10), pages 77–96, Montreal, Quebec, Canada, Oct. 2007.
- [44] 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 Symposium on Operating Systems Design and Implementation*, pages 103–116, Seattle, WA, Nov. 2006.
- [45] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic heap sizing: taking real memory into account. In *Proceedings of the International Symposium on Memory Management*, pages 61–72, Vancouver, BC, Canada, June 2004.
- [46] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara. Program-level adaptive memory management. In *Proceedings of the International Symposium on Memory Management*, pages 174–183, Ottawa, ON, Canada, June 2006.
- [47] Y. Zhang, A. Bestavros, M. Guirguis, I. Matta, and R. West. Friendly virtual machines: leveraging a feedback-control model for application adaptation. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 2–12, Chicago, IL, June 2005.
- [48] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 39(11), pages 177–188, Boston, MA, Oct. 2004.