Waste Not, Want Not
Resource-based Garbage Collection in a Shared Environment

Matthew Hertz, Jonathan Bard, Stephen Kane, Elizabeth Keudel
{hertzm,bardj,kane8,keudele}@canisius.edu
Canisius College
Department of Computer Science

Tongxin Bai, Kirk Kelsey, and Chen Ding
{bai,kelsey,ding}@cs.rochester.edu
The University of Rochester
Computer Science Department
Rochester, NY 14627

Technical Report TR-951
December 2009

The research is supported by the National Science Foundation (Contract No. CNS-0834566, CNS-0720796), IBM CAS Faculty Fellowship, and a gift from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding organizations.
Abstract

While a conventional program uses exactly as much memory as it needs, the memory use of a garbage-collected program can be adjusted by changing the size of the heap used by the garbage collector. This difference can allow applications to adapt their memory demands in response to the changing amount of available memory in a shared environment, which is increasingly important for today’s multicore, multiprocessor machines.

We present a memory performance model that better addresses issues that occur with the more changeable memory demands of garbage-collected applications. Traditional locality models are not directly applicable because the application demand may change based on the available memory size. We describe time-memory curves, which can be used to derive optimal static memory allocation in a shared environment. For dynamic environments, however, more will be needed. In this work, we describe Poor Richard’s Memory Manager, a lightweight system to reduce paging costs that can be incorporated into existing runtime systems. We describe the design of the memory manager and show how it can be added to most existing garbage collectors with little to no effort. Using an experimental evaluation of both homogeneous and heterogeneous Java workloads on a dual processor machine, we show that Poor Richard’s Memory Manager improves average performance by a factor of 3 or more when paging, while adding almost no overhead when there is no memory pressure. We further show that this system is not specific to any single algorithm, but improves every garbage collector on which it is tested. We finally demonstrate the versatility of our memory manager by using it to improve the performance of a range of .Net workloads.
1 Introduction

Memory usage depends first on program demand. Traditionally in programs that do not use dynamic memory allocation, such as those written in Fortran 77, and those that use explicit memory de-allocation, such as standard C and C++ programs, the memory demand depends completely on program factors.

Today’s software developers are increasingly taking advantage of garbage collection for the many engineering benefits it provides using either garbage-collected languages such as Lisp, ML, and Java or conservative garbage collectors such as the Boehm-Demers-Weiser collector [6]. The memory use of a garbage collected program often expands beyond the minimum need of the program to take advantage of all available memory in the system, an idea we refer to as resource-based garbage collection.

A garbage-collected program’s heap must be large enough to accommodate all reachable data. If more memory is available, the virtual machine can grow the heap. The larger heap size leads to fewer calls to the collector, often resulting in reduced collection times. But if the heap grows too large, the heap can exceed the available memory and trigger paging and greatly reduce performance.

The common scheme for Java programs is to use a fixed range of heap sizes which often exceed the program’s need. As an example, the Jikes RVM defaults to a range of 50MB to 100MB. This preset range is often a mismatch with the resources available for a number of reasons. First, the actual memory usage depends on the garbage collector implementation (e.g. copying collector uses twice as much as the reachable data), and does not account for other memory usage, such as that for the virtual machine and operating system. Second, knowing the exact amount of available memory is difficult because it requires ascertaining the actual memory usage of other programs. Some memory may be occupied but not used (for example, by file caches) and can be made available. Last but not least, this preset heap size cannot respond to change of conditions in the middle of an execution.

Previous work has studied resource-based garbage collection in an exclusive environment, in which only one program is expanding its memory usage to occupy the available memory. In this paper we address the problem of shared resource utilization, in which a group of programs share the same memory resource.

Memory sharing is increasingly common on today’s multi-processor, multi-core, and multi-threaded machines. The traditional way of manually specifying the heap size becomes more problematic. Since the system load may change dynamically and unpredictably, being conservative may leave most of the system memory unused, while being aggressive may lead to severe contention. Resource-based solutions should help. However, the moment a program starts to adapt, it embarks on a collision course with other like minded programs, no matter how much memory is available.

We first revise the classical performance model, which is demand-based, to accommodate resource-based memory management. We introduce a new quantitative notion called time-memory curve and use it to predict the performance of memory sharing in this new context.

Next we describe a resource-based memory management system called Poor Richard’s Memory Manager. Poor Richard’s improves performance by monitoring the application’s behavior, and sometimes that of other applications, to enable the program to adjust its memory demands dynamically and work within the current level of available resources.
Given the shared environment, Poor Richard’s Memory Manager usually attempts to have applications communicate and work cooperatively to reduce memory pressure. Communication is done simply using a small amount of shared memory, that we call the *whiteboard*. Using this *whiteboard*, applications can alert one another when paging occurs and ensure the affiliated processes reduce their memory demands. When the system is paging, cooperative PAMM also serializes any elective garbage collections to limit the increased memory demands that occur when collecting the heap. We also present a *selfish* scheme in which each process strives to maximize its own memory usage without regard to other applications. Selfish processes thus avoid the overhead needed to communicate between jobs. The tradeoffs of increased cooperation and reduced overhead are especially important on a multi-core, multi-processor environment where applications are constantly running and allocating memory.

Because Poor Richard’s Memory Manager requires only limited interaction between the memory manager and virtual machine its implementation is entirely independent of the garbage collector. We needed under 200 LOC to add support for Poor Richard’s Memory Manager to all of the generational collectors in the Jikes RVM [4] and did not need to make any changes to our Memory Manager. Using common Java benchmarks, including SPECjbb and selections from the DaCapo suite, we evaluate both the cooperative and selfish scheme of Poor Richard’s Memory Manager. As we will show, this simplicity allows our system to not degrade performance when memory pressure is low and improve performance by up to a factor of 3 as memory pressure increases. We will also show that performance improvements can be seen in every collector we tests. To show the versatility of our system, we then added Poor Richard’s Memory Manager to the conservative, whole-heap Boehm-Demers-Weiser collector [6] used in Mono, the open-source .Net runtime system. This required that we add under 100 LOC to the Mono system and still did not require that we modify our code. As with the generational collectors, Poor Richard’s Memory Manager was able to offer substantial performance improvement to the Mono system when paging.

2 Memory performance model

Intuitively more memory leads to better performance. The goal of the memory-performance model is to show what amount of memory can improve performance for which programs and by how much. The application-specific part of the model is also referred to as the locality model of an application.

2.1 The classical model

The classical model of memory performance as pioneered by work of Mattson et al., Denning, and Smith [10, 18, 22] is presented as a lifetime curve [10] or a miss-rate curve for page and cache access [18,22]. The x-axis of these curves shows the amount of available memory. The y-axis shows the performance, which in the case of a life-time curve is the average time between two page faults and in a miss-rate curve the average number of faults per 100 page or memory accesses. We refer to these models collectively as memory performance curves. They provide the exact relationship, so we can partition shared memory appropriately, for example, parceling out shared memory to maximize the overall performance gain or to equalize individual performance gains.

One basic assumption made in these classical models is that a program’s memory demand does not change with the environment. The program has a different performance when running with
different amounts of memory but it is always the same program demand. Consider an analogy that a program is a ball in a room, the amount of memory is the ambient light, and the performance is the shadow of the ball determined jointly by the ball and the light. The analogous assumption in the classical model is that while the shadow changes with the change in the ambient light, the ball itself does not change.

2.2 The effect of heap size

For programs that use garbage collection, memory demand depends on the heap management in the virtual machine, in particular, the size of program heap. In current systems, a default heap size, $H_{\text{default}}$, is used, and a garbage collector is invoked when the heap is full. Part of the default scheme is a provision for increasing the heap limit if it is too small to hold the reachable data needed by the program. Later we will refer to this as demand-based heap growth.

Starting a program with different default heap sizes, $H_{\text{default}}$, changes memory utilization and program performance. The best performance happens when the heap size is such that the effective heap use is equal to the size of available memory. If the heap is too small, a program does not use all the available memory and spends more time than necessary in garbage collection. If the heap is too large, the program incurs extra cost of paging.

The assumption in the classical model is invalid for these programs because their demand changes when $H_{\text{default}}$ changes. In the ball-in-a-room analogy, these programs are stretchable balls and can change size by itself and consequently its shadow with or without a concurring change in its environment. The memory performance depends on two orthogonal factors, heap size and memory size. In contrast, the performance of traditional programs depends on only memory size.

We show the effect of heap size through measurement of a Java program $ipsxql$ when running with a different amount of physical memory. Figure 1 shows a 3-D plot. The $z$-axis shows the running time, $y$-axis shows the total physical memory in the system, and $x$-axis shows the default heap size used for each execution.

The figure shows a significant effect from the default heap size. The valley in the graph around a heap size of 100M represents a good choice for those physical memory sizes. If the heap size deviates from this value, the results degrade rapidly. For larger physical memory sizes, the valley plains out and the choice of heap size becomes somewhat less critical.

The effect has two important implications for a shared environment. First, as the available memory changes dynamically in a shared environment, a fixed heap size may severely under- or over-utilize available memory and may incur an extremely high penalty. Second, even for an adaptive scheme that can find the best heap size, a parallel execution may not improve the throughput when there is not enough memory to share. For example for the memory size 112MB, the program with the best fixed heap size takes almost three times as long as the program would with twice the memory. Therefore, running two copies sequentially using one processor is 40% faster than running the same two copies in parallel using two processors.
2.3 The time-memory curve

For all heap sizes of a program, if we take the best performance for each physical memory size, we obtain a curve we call *time-memory curve*, which shows how performance depends on the amount of available memory.

**Definition 1 (Time-memory curve)** The time-memory curve of a garbage-collected program is a function $t(m)$. For each size of available memory $m$, $t(m)$ is the lowest running time possible among all heap size choices.

By taking the optimal heap size, time-memory curve has similar dimensions as the classical model and can have as many uses for GC-based programs as the classical model for traditional programs, provided that we find a way to measure the time-memory curve for a program.

In this paper, we use resource-based heap management. The idea is to monitor the heap usage and grow the heap limit until it uses all available memory. As a result, the heap size depends on the size of the available memory in the system rather than a fixed starting point like $H_{\text{default}}$. Two techniques have been developed based on this idea: *program-level adaptive memory management (PAMM)* ([26], also described in Section 4) and Isla Vista heap sizing by Grzegorczyk et al. [14]

They manage heap with resource-based growth as opposed to the demand-based growth in the conventional setup. Resource-based memory management has its limitations and may not always achieve the best possible performance. Our earlier work has compared its performance with the performance of the best fixed heap size [26]. PAMM for the most times achieves the best observed performance, but it is still an approximation since we do not have a way to optimize the performance for a given memory size.

---

1. Resource-based growth becomes the same as demand-based growth if the size of live data exceeds the size of available memory.
How does the performance change as a function of memory usage? We show typical variations in Figure 2 using the time-memory curve of four SPEC JVM98 benchmarks, *jess*, *javac*, *compress*, and *mtrt*, measured using a version of PAMM. As the amount of physical memory changes from 64MB to 128MB, the running time changes from 278 seconds to 2.4 seconds. Most changes happen at the left end of the figure, between 64MB and 96MB memory. To better show the smaller variations between execution times obtained using larger memory sizes, we draw the $y$-axis using a logarithmic scale (base 2).

All curves in Figure 2 show a “knee”, which is well known in the literature of virtual memory management as the working set [10]. The non-linear nature is the result of locality—that programs use only a subset of their data most times. The time-memory curve quantifies the locality of garbage collected programs and, as we will discuss in Section 3, has uses in memory sharing for garbage-collected programs as an extension of the working set model. Furthermore, the extension enables GC-based programs and normal programs to be managed together to effectively share memory.

**Resource-based GC versus exhaustive testing** Using PAMM or Isla Vista to approximate the time-memory curve has several practical benefits. First, the measurement takes one execution for each physical memory size. It is more efficient than searching through all possible heap sizes. On the other hand, measuring the time-memory curve is more costly than building the classical model, which can be done in one-pass (for inclusive caching) for all memory sizes. Resource-based heap growth violates the inclusion principle because its memory management depends on the memory size (see pp. 93 of [18]). On the plus side, the performance of resource-based memory management
is measured by running the application without the high cost of simulation. In addition, the result shows performance directly, while simulation of reuse or stack distances is indirect.

Second and more importantly, PAMM and Isla Vista enable a program to adapt its heap size to behave according to the time-memory model. This is important for two reasons. First, the actual amount of available memory is often unknown. Second, the available memory changes dynamically in a shared environment. Resource-based heap growth dynamically changes the size of the heap to utilize available memory.

Alternatively we can approximate the time-memory curve through exhaustive search. If we test all fixed heap sizes for each physical memory size as we have done in the experiment reported in Figure 1, taking the minimal time for each memory size is the same as finding the lowest point along the heap-size dimension. The result is the valley in a 2-D plot.

While it is possible for a user to test and find the best fixed heap size, the manual control is difficult for three reasons. First, the actual memory demand depends on the garbage collector and may be much more than $H_{default}$. Second, finding the exact amount of available memory requires ascertaining the active memory usage of the operating system, the virtual machine, and other concurrent programs. Finally, $H_{default}$ is set at the beginning and cannot respond to a change of conditions in the middle of an execution. While in theory a user can choose a good $H_{default}$ knowing the physical memory size on a machine, in reality the task is beyond the grasp of all but the most proficient and ardent users. Sometimes it happens that the best fixed heap size does not perform as effectively as the adaptive scheme because the latter may use different heap sizes in the same run [26]. To find the minimal $t_{default}$, in the worst case, a manual or automatic search needs to check every $h_{default}$, which takes $O(M)$ number of runs for each memory size $M$.

3 Static memory management for a shared environment

In a shared environment, the throughput, response time, and fairness depend on the memory allocation among concurrent applications. We consider a simplified problem where we run a set of programs on a multi-processor and we want to maximize throughput, that is, to finish all programs in the shortest amount of time. We assume that the machine has as many processors as the number of programs, so the chief question is how to make best use of the memory resource.

3.1 To share or not to share

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

**Proposition 1 (Concurrent execution)** A concurrent execution for $N$ programs running on $N$ identical processors with a total amount of memory $M$ finishes no slower than running those $N$ programs sequentially if

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

for $\exists m_1, m_2, \ldots, m_n$ such that

$$\max_{i=1\ldots N} t_i(m_i) \leq \sum_{i=1}^{N} t_i(M)$$
where \( t_i \) is the time-memory curve of program \( i \) and \( \sum_{i=1}^{N} m_i \leq M \).

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

\[
t(M/N) \leq Nt(M)
\]

Take for example \( jess \) shown in Figure 2. Suppose we want to complete two runs of the program on a machine with 128MB memory. If we run the two in parallel, the optimal memory allocation, which we will derive shortly, is for each instance to use 64MB of memory and to finish in a total of 278 seconds. If we run them sequentially, each uses the entire 128MB memory and finishes in 2.5 seconds, so the total sequential time is 5 seconds, a factor of 55 lower than the parallel time. In other words, sharing is detrimental when there is insufficient memory.

The threshold memory size for profitable parallel execution (in terms of throughput) is given by the corollary. The performance of \( jess \) stays at 2.5 seconds with 128MB or more memory. For parallel execution to finish faster at 64MB per \( jess \) instance, we would need to run 56 instances before the parallel time 278 seconds becomes lower than the sequential time of 56 times 2.5 or 280 seconds, both using 56 times 64MB memory. If we have 80MB per \( jess \), the parallel execution, 8.7 seconds according to time-memory result, would gain an upper hand with 4 instances with a total of 320MB memory, compared to the sequential execution using the same amount of memory.

A similar calculation can show that the threshold memory size for two parallel tasks is 192MB for \( jess, mrt, \) and \( compress \), with parallel time being 44%, 30%, and 5% shorter than the sequential time.

This model does not consider the sharing of a single processor because the overlap of computation and paging depends on the program and the file system. We also assume no I/O contention from paging of multiple processes. Despite these limitations, the time-memory curve can be a useful indicator for predicting the effect of memory sharing and can be used to find the multiprogramming scheme with the best throughput.

**The memory benefit of parallel processing**  Parallel processing usually implies an increase in memory pressure. Our calculation here turns this relation in reverse direction—extra processors can be viewed as memory amplifiers since they enable performance equivalent to single processor with a larger amount of memory. To compute this we adapt the formula of the corollary of Proposition 1 to

\[
t(M/N) \leq Nt(MN)
\]

For \( compress \), when \( M = 192 \) and \( N = 2 \), \( t(M/N) = 8.7 \leq Nt(MN) = 9.2 \). Therefore, an extra processor has the equivalent benefit of 192MB memory.

### 3.2 Optimal static sharing

With the time-memory curve, we can derive the optimal memory allocation for a set of programs. Since time-memory curves are monotonic, a greedy solution suffices. The basic idea is to start each program with the minimal amount of memory required (the start of its time-memory curve) and then
keep allocating the next chunk to the program that benefits most from new memory. The benefit is measured by the execution time reduction per unit of memory. Visually, at each increment of memory allocation, we choose the curve that has the deepest descent. We call the algorithm greedy descent allocation.

Take for example the problem of allocating memory for the parallel execution of jess and mttrs. We assume that the executions are infinitely long, and we compute the throughput normalized to the throughput of the base case at 64MB memory. The total throughput at 128MB memory (64MB each) is 2. In the time-memory curve shown in Figure 2, additional 16MB memory reduces jess time from 278 seconds to 8.7 seconds and compress time from 122 seconds to 15.6 seconds. The throughput improvement is a factor of 32 for jess and 7.8 for mttrs. Greedy descent chooses the larger factor and allocates the 16MB to jess to improve the total normalized throughput from 2 to 33. Greedy descent allocates the next 16MB to mttrs because the next improvement of jess is a factor of 3 (8.7 seconds down to 2.8 seconds in the time-memory curve), which is less than 7.8 of mttrs. Therefore the best throughput from a fixed allocation of 160MB total memory is 39.8. We call it statically optimal.

The preceding calculation can be used for any discrete form of time-memory curve, where the performance is measured on any subset of memory sizes. Measuring a subset of memory sizes does not show complete time-memory performance, but the solution of greedy descent can make use partial time-memory information as just demonstrated.

The time-memory model can be used to derive theoretical properties similar to the ones in the traditional literature of virtual memory management. For example, when allocating memory among programs with identical time-memory curves, the intuitive solution is equal partition. Greedy descent shows the exact condition for the intuitive solution to be optimal, as given by the next proposition.

**Proposition 2 (Optimality of equal allocation)** For programs with identical time-memory curves, equal memory allocation maximizes the parallel performance if and only if the gradient of the curve is no non-increasing.

Denning defined the primary and secondary working sets as the global and local maximal gradient. The proposition states that equal allocation is optimal for programs with a single working set but not optimal for programs with multiple working sets, that is, multiple “knees” in the time-memory curve.

As the number of processors increases on modern machines, memory is increasingly the bottleneck. The new model of time-memory curve is useful for deciding the best memory allocation in a shared environment. It is an important for data-intensive applications for which memory is a more critical resource than CPU is. It requires good models because the relation between performance and available memory is non-linear because of locality.

**Limitations** The optimality is for static sharing, that is, the best allocation for the entire execution. Static sharing as presented has three serious limitations. First, it requires the time-memory curve before the execution, which is difficult for complex code whose behavior may depend on (run-time) inputs. Second, the curve shows the average behavior, while the temporal behavior may change significantly from time to time. Finally, the environment, in particular the amount of available memory,
may change due to dynamic arrival and departure of other programs. In fact, the theoretically optimal partition for *jess* and *compress* can be easily improved in reality. The two programs do not finish at the same time, and we can make the longer-running one finish faster by letting it use the entire memory instead of its allotted share. Dynamic memory sharing will address these limitations, as we discuss next.

4 Poor Richard’s Memory Manager

We begin by discussing the development of the *cooperative* version of Poor Richard’s Memory Manager. We then discuss the ideas and reasons behind our implementing a *selfish* version of Poor Richard. We finally show how Poor Richard’s limited interactions with the virtual machine and garbage collector and leads it to be nearly independent of the language, garbage collector, and virtual machine. Given this independence we discuss how we can add Poor Richard to any garbage collector and system.

4.1 Penny Saved Is a Penny Earned

At the outset, the goal of Poor Richard’s Memory Manager was to capitalize on the already good performance of existing virtual machines and garbage collectors when not paging and improve the far less good performance when paging. To achieve this goal, we minimized the frequency with which Poor Richard interacts with a garbage collector and information shared during each interaction. Systems already include a “slow-path” check of whether a (demand-driven) collection is needed. We added code so that the slow-path code periodically polls Poor Richard’s controller to see if a resource-driven collection is needed. This call to the controller is additive only and will not interfere with any demand-driven need to collect the heap. If resources are plentiful, the controller would not signal the need for a collection, and the performance of the existing system is undisturbed. Figure 3(a) shows an example of the changes that would be needed to support Poor Richard’s Memory Manager.

The simplicity of this interaction has other benefits. The changes do not depend on knowing the specifics of any GC algorithm, only on where these slow-path checks occur. By adding this code to the code used by all the generational collector in the Jikes RVM, we were able to add support for Poor Richard’s Memory Manager for all of these collectors in fewer than 200 LOC. This simplicity meant that fewer than 100 LOC were needed to add our system to the open-source .Net runtime system, Mono. As we will now show, we did not need to make any changes to the code implementing Poor Richard’s Memory Manager to move between supporting these two system.

For each process to evaluate their own level of memory pressure, we can rely on the `/proc/self/stat` and `/proc/self/statm` pseudo-files provided by the Linux operating system. As part of the `/proc` file system, neither of these files really exists, but instead serve as interfaces to the data structured maintained in memory by the kernel.

Checking these files could entail non-trivial costs. We reduce this cost in two ways. First, the entire controller, including the reading and parsing of files, is written in C. For systems implemented in Java, such as the Jikes RVM, the call to our controller uses the Java native interface (JNI). Second, we limit the frequency with which these checks are performed to once every 100 times the slow-path
(a) Modified slow-path allocation code

```java
public static boolean USING_PRMM = true;
public final boolean gcCheck() {
    int nurseryPages = nurserySpace.reservedPages();
    if (nurseryPages > Options.nurserySize.getMaxNursery() ||
        nurseryPages >= getMaturePhysicalPagesAvail())
        return true;
    if (USING_PRMM && ++numAllocs >= 100) {
        numAllocs = 0;
        int result = consultPoorRichard();
        if (result != 0) {
            forceFullHeapCollection = (result == 1);
            return true;
        }
    }
    return false;
}
```

(b) Poor Richard’s Memory Manager code checking memory pressure

```c
extern "C" int consultPoorRichard() {
    long long currentPFaults = getPageFaults();
    long currentRSS = getResidentSetSize();
    return whiteboardCheck(currentPFaults, currentRSS);
}
```

(c) Whiteboard used to communicate memory pressure and serialize GC

```c
int whiteboardCheck(long long faults, long rss) {
    gcType = NO_GC;
    pthread_mutex_lock(&whiteboard->mutex);
    for (int i = whiteboard->entries; i >= 0; i--) {
        if (whiteboard->processes[i].pid == pid) {
            if (whiteboard->processes[i].flag) {
                gcType = TRIGGER_LIMITED_GC;
                break;
            }
            else if (rss < whiteboard->processes[i].rssSize ||
                     faults <= whiteboard->processes[i].fault < 10) {
                gcType = TRIGGER_LIMITED_GC;
                setFlags(); // set all processes[i].flag fields */
                break;
            }
            if (gcType == TRIGGER_LIMITED_GC && !whiteboard->inGC) {
                whiteboard->inGC = 1; /* set exclusive GC right */
                gcType = RESOURCE_TRIGGER_GC;
            }
            pthread_mutex_unlock(&whiteboard->mutex);
            return gcType;
        }
    }
}
```

Figure 3: Poor Richard’s Memory Manager code
is executed. This constant rate is one of the two parameters within the design of Poor Richard, but is hard-coded into the virtual machine during its compilation. We discuss the use of these constants during our presentation of results in the following section.

4.2 “Little Strokes Fell Great Oaks”

In a shared environment, Poor Richard’s Memory Manager has the shortcoming that all processes may detect paging activities and start collecting their heaps at the same time. Because the act of collecting the heap temporarily increases a processes memory demands, this sudden onset of clustered GC activities may increase paging demands and cause severe contention for the disk. An early goal of this work is for parallel processes to coordinate their activities so that paging problems could quickly be disseminated and the collections needed to relieve this memory pressure serialized.

In this cooperative approach, applications share a common whiteboard on which they can record their page fault counts, resident set sizes, and if they are currently in a resource-driven collection of their heap. This cam then be used by the Poor Richard controller to determine if a resource-driven collection is necessary. The function consultPoorRichard in Figure 3(b) shows the data each process collects before using the whiteboard to determine if a collection is needed.

The whiteboard itself is a small amount of shared memory that is used by all of the cooperating applications. The whiteboardCheck function is shown in Figure 3(c). After finding its record in the whiteboard, there are three separate triggers we use to determine if memory pressure exists. The first branch checks if a separate process has alerted the process that it is having paging issues and that all cooperating processes should reduce their memory demands. The second branch checks to see if the current process is, or will soon be, paging. The first test in this branch checks to see if the process’s resident set size (RSS) has decreased since the last GC. The resident set size measures the number of pages that are actually in memory; a decrease occurs only if the process has had pages evicted. The second test checks if the number of pages faults since the last GC has exceeded some threshold value. This threshold value, 10 in the experiments presented in this paper, is the second parameter within Poor Richard’s Memory Manager and is also hard-coded into the system during compilation. If the controller finds that either condition has been met, it will trigger the need for a resource-driven collection of the system. Since the controller compares both the page fault count and RSS with those values recorded after the last GC, the frequency with which the Poor Richard controller is polled does not change the decisions made.

If multiple cooperative applications began collecting their heap at the same time, their increased memory demands could result in increased paging and contention accessing the disk. We prevent this by having systems use the whiteboard to record whenever they are performing a resource-driven collection. At the start of this collection, Poor Richard’s Memory Manager sets whiteboard->inGC. While this field is set, no other process will trigger a resource-driven collection, though demand-driven collections will be allowed. When a process completes a resource-driven collection, it will then clear the whiteboard->inGC flag. Once clear, other processes can then perform their resource-driven collection. This results in these collections being serialized and prevents the system from becoming overloaded by clustered collections.
Figure 4: “Selfish” runs of Poor Richard use a simpler check for memory pressure that does not use the whiteboard

4.3 “The Lord Helps Those Who Help Themselves”

We also implemented a selfish version of Poor Richard’s Memory Manager. This selfish scheme eliminates the locking and communication overhead needed by the cooperative system, but also cannot prevent multiple processes from having their resource-driven collections clustered in time. Both cooperative and selfish runs of Poor Richard require the same support from the virtual machine, so the polling process shown in Figure 3(a) remains. When polled, the selfish system continues to track the page fault and resident set size data found in the /proc/self/stat and /proc/self/statm pseudo-files. As the code in Figure 4, the selfish version of our system continues to trigger a collection when the number of page faults since the last collection is larger than our threshold value (for these systems we also used a threshold of 10) or the resident set size drops.

5 Results

We now present results analyzing the performance of the cooperative and selfish versions of Poor Richard’s Memory Manager and compare this performance with the default approach within the Jikes RVM. We also discuss how we could implement Poor Richard’s using a variety garbage collectors and show results from those collectors. Finally, we discuss the porting of Poor Richard’s Memory Manager to the Mono runtime and present results from this system. We first describe the collectors we will use and the benchmarks on which we run our collectors in more detail below.

5.1 Methodology

We performed our experiments on a dual-processor Linux machine. Each processor is a single-core, hyperthreaded 2.8GHz Intel Xeon processor with 1MB of L2 cache. We use grub loader configurations to limit the amount of physical memory to used. For these experiments, we limited the physical memory to 256MB. The machine was placed in single-user mode with all but the necessary processes stopped and the network disables. All experiments were performed by running two
applications concurrently and we report the time required before both processes were complete. Each data point represents the mean of 5 set of runs for the given heap size.

For our experiments, we chose two set of benchmarks. The first benchmark was pseudoJBB, a fixed workload variant of SPECjbb [9]. The second set of benchmarks we used were the four benchmarks from the 08-2006-beta release of the DaCapo benchmark suite that ran in the JVM: bloat, fop, pmd, and xalan [13].

It has been shown previously that compilation costs can bias results of experiments [12]. As a result, most experiments follow a second run methodology [3]. We observed, however, that the first compilation pass can trigger significant paging which influences the results of further passes. Since we cannot control or eliminate this effect, we chose to measure at least 5 runs of each benchmark to minimize the bias of the compilation costs.

Using these benchmarks we ran two sets of experiments. The first set of experiments examined homogeneous workloads. For these workloads, we measured the time needed to execute two parallel instances of each benchmark. For our second set of experiments, we examined heterogeneous workloads. In these experiments we ran each possible pairing of the DaCapo benchmarks. While each of these benchmarks would normally complete in a different amount of time, we selected the number of passes to run each benchmark so that they would complete within approximately one second of each other when run with no memory pressure. Table 1 provides important metrics for each of these benchmarks.

For all of our Java experiments, we used the Jikes RVM (version 3.0.1, release 15128). To limit variations between runs, these experiments used a pseudo-adaptive compilation methodology [16, 20]. Under this approach, we first timed 5 separate runs of each benchmark using the adaptive compiler and recording the final optimization decisions made in each run. Using the decisions from the fastest of these 5 runs, all of our experiments duplicate the decisions of an adaptive compiler strategy but in an entirely repeatable manner.

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

Table 1: Key Metrics for Single Pass of Each Benchmark

5.2 “Early Bird Catches The Worm”

These initial results are for the default collector used by the Jikes RVM, a generational collector which uses a mark-sweep policy to collect its mature space (“GenMS”). This is the default collector on the Jikes RVM and provides the best average performance. Figures 5 and 6 compares the

We also examined the results if we used the average completion time of each process and found this did not change our relative results or conclusions.
Figure 5: Graphs showing that Poor Richard’s Memory Manager improves the paging performance of the GenMS collector in Jikes RVM for both homogeneous and heterogeneous workloads. Because paging’s impact seems to be felt universally, the selfish version of the system will meet or exceed the cooperative version of the system.
(a) Average performance across all experiments of the default collector and the collector using selfish and cooperative versions of Poor Richard

(b) Average performance across all heterogeneous experiments after breaking out the performance of each trigger used by Poor Richard

Figure 6: Graphs showing that Poor Richard's Memory Manager improves the paging performance of the GenMS collector in Jikes RVM for both homogeneous and heterogeneous workloads. These graphs show that using both the resident set size and page fault counts to trigger collection exceed the performance of either trigger by itself.
performance of the default Jikes RVM with runs of the Jikes RVM to which we added either the cooperative or selfish versions of Poor Richard’s Memory Manager. We further breakdown results for Poor Richard’s Memory Manager to consider its performance with and without the resident set size trigger. In each of these graphs, the x-axis shows the fixed heap size specified for the system when each run began and the y-axis shows the execution time needed at each heap size relative to the time needed by the default cooperative version of Poor Richard’s Memory Manager using both triggers. In each of these graphs, the results for all 5 systems are largely the same at the smallest heap sizes when there is no memory pressure. In all of our experiments, the results for the systems at this smallest heap size was always within 5% of each other. For the default collector and the runs to which we added either the cooperative or selfish memory manager using both the page fault and resident set size triggers (“GenMS + Coop + RSS” and “GenMS + Selfish + RSS”), the results were always within 1% of each other. This shows that we met our first goal of developing a lightweight approach that did not interfere with the already good performance of existing collectors. It also suggests that the locking overhead required for the cooperative systems is not too high.

As soon as paging begins, the benefits of our approach become immediately apparent. In Figure 5(a), we can see that the performance of the default collector quickly degrades as a result of paging. The systems augmented with Poor Richard’s, however, see little change in their performance. Across all of the different heap sizes in this figure, the average execution times for both the selfish and cooperative systems using both triggers differ by less than 10%. Figure 5(b) shows that our system also improves performance when running pseudoJBB in parallel. While this experiment was the worst that Poor Richard’s performed relative to GenMS, the default system still ran at least 10% slower when paging and was often between 30% and 70% slower. The overall improvements made possible by Poor Richard’s Memory Manager are best shown in Figure 6(a). As this shows, with only a very small addition to the Jikes RVM we were able to reduce the time needed to execute two parallel runs by at least 20%. In most cases, we averaged between a factor of 2 to nearly factor of 3 improvement over the default system.

Interestingly, these results also show that the selfish version of the system consistently outperforms the cooperative version of Poor Richard’s Memory Manager. As we can see in Figure 5(a), the selfish approach does not always improve performance. As in that example, we found that often the two approaches were equally good at limiting the performance degradation from paging. As the results in Figure 6(a) suggest, however, when there were differences, the selfish controller would need up to one-half the time required for the cooperative controller. The reason for this difference was surprising: it was that the cooperative systems could NOT cluster their collections. While this could be a problem if the system were paging heavily, both approaches kept paging to an absolute minimum. As a result, the risk that clustering collections leads to disk contention is much smaller than the increased benefit of reducing the memory pressure that much quicker.

We also investigated whether we needed both the page fault count and the resident set size to achieve this good performance. The graphs in Figures 5 and 6 include results for Poor Richard’s Memory Manager using just page faults and both page faults and the resident set size as triggers. As these graphs show, adding the resident set size as a trigger provides a sizeable performance improvement over just using the page fault count. The resident set size drops when pages are evicted from memory onto the disk. As a result, a drop in the resident set size is a clear indication of both increased memory pressure and of future page faults when the program next touches the evicted pages. The resident set size trigger therefore warns an application even earlier that it will
soon have problems from paging. By collecting the heap sooner, Poor Richard’s Memory Manager can avoid the more significant paging events in the future.

We therefore wondered if it would be possible to only use the resident set size trigger. Figure 6(b) shows the average (geometric mean) result of using just an RSS trigger on the experiments for heterogeneous workloads. As the graph clearly shows, this is not a good idea. The problem is that as the heap grows and new pages are allocated, the resident set size grows. So long as the application is able to allocate more pages than are evicted, Poor Richard’s would not detect the increased memory pressure and cause the application to collect and shrink its heap. This leads to the poor performance we can see in this graph and the need for Poor Richard’s dual trigger.

5.3 “Only Certainties in life are death, taxes, and Poor Richard”

We also investigated whether Poor Richard’s Memory Manager would be able to improve the performance of other garbage collectors. Because of Jikes RVM and MMTk’s open design, we were able to add the code needed to poll Poor Richard’s controller to all of the generational collectors with a single modification. While the other collectors either were still experimental or do not perform as well as GenMS, we wanted to see if our approach really was universal. Figure 7(a) shows that, without any modifications, our configuration of Poor Richard’s Memory Manager was able to improve the performance of a generational collector using a copying mature space (“GenCopy”) by a speedup factor of 2 versus the default GenCopy collector. While the improvements were not as great Figure 7(b), our unmodified Poor Richard’s system was also able to improve runs using an Immix space [5] to collect the mature objects. These results show that the general nature of Poor Richard’s Memory Manager makes it easy to use and improve the paging performance of any collector.

As a last experiment to see how adaptable our approach really was, we also ported Poor Richard’s to Mono [19]. This port was interesting because the garbage collector used by Mono, a conservative, whole-heap collector based upon the BDW collector, is very different from the one on which he previous performed our tests. The differences were not hard to overcome, however. This port required adding under 10 lines of code to the Mono code base and did not require any changes to our memory manager. Much more difficult was finding benchmarks we could run on this system that were capable of generating any heap that could trigger paging. In the end, we relied on a port of the GCOld synthetic benchmark which we tested using several different ratios of short-to-long lived objects. Figure 8 shows that Poor Richard’s Memory Manager was even able to improve the paging performance of this system, offering a speedup between 1.5 and 1.73.

6 Related work

The prevailing models of memory performance assume that an application does not change its memory demand based on the available memory [10,18,22]. Therefore they cannot predict performance of resource-based adaptive memory management in either single or multi-programming environments. We present the time-memory curve as a solution and show its use in understanding the effect of memory allocation in a shared environment.
(a) Average performance across all heterogeneous experiments of generational collectors using mark-sweep and copying collection for the mature objects. These results also show the result of adding cooperative versions of Poor Richard to both collectors.

(b) Performance of generational collectors using mark-sweep and immix collection for the mature objects running bloat and fop in parallel. These results also show the result of adding cooperative versions of Poor Richard to both collectors.

Figure 7: Graphs showing that Poor Richard’s Memory Manager improves the paging performance of all generational collectors in the Jikes RVM.
Resource-based memory management has been studied originally at the level of the virtual machine. Alonso and Appel presented a collector which reduced the heap size when advised that memory pressure was increasing [1]. Yang et al. modified the operating system to use an approximate reuse distance histogram to estimate the current available memory size. They then developed collector models enabling the JVM to select a heap size that fully utilize physical memory [24, 25]. Instead of changing memory allocation, Hertz et al. developed a paging-aware garbage collector and modified virtual memory manager that cooperated to greatly reduce the paging costs of large heaps [15].

Recently, Grzegorczyk et al. examined the causes of paging based on the virtual memory implementation of a Linux operating system and found that another OS event, page allocation stall, was a better indicator of memory pressure than the page fault event [14]. On our system, the frequency of page faults is one order of magnitude higher than the frequency of allocation stall. We use page faults for eager detection, which is beneficial in a shared environment. We are evaluating the use of allocation stall.

The past schemes do not specifically consider memory sharing by multiple JVMs. This is the focus of the current study.

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 [2]. The average performance improvement for SPECjbb2K running on JRocket was 2% with the learning overhead and 6% otherwise. Instead of using a fixed cost and memory size, our recent work adaptively monitored the number of page faults and adjusted the heap size of a program in an exclusive environment [26]. Both of the methods required manual analysis of the program. In addition to the formal modeling, our work complements the past program- and VM-based techniques in two aspects. First, we show that run-time monitoring can be fully automated and can therefore be applied to general programs. More importantly, we develop a cooperative scheme for use of resource-based adaptation in a shared environment.
Many other adaptive schemes have been used for garbage collection. Several recent studies examined adaptation based on the program demand. Buytaert et al. use offline profiling to determine the amount of reachable data as the program runs and generate a listing of program points when collecting the heap will be most favorable. At runtime, they then can then collect the heap when the ratio of reachable to unreachable data is most effective [7]. Similar work by Ding et al. used a Lisp interpreter to show that limiting collections to occur only at phase boundaries reduced GC overhead and improved data locality [11]. Soman et al. used profiling, user annotation, and a modified JVM so a program may select which garbage collector to use at the program loading time [23]. Our work has orthogonal purposes because it automatically adapts to the changing resources at run time.

While heap management adds several new wrinkles, there has long been work on creating virtual memory managers which adapt to program behavior to reduce paging. Smaragdakis et al. developed early eviction LRU (EELRU), which made use of recency information to improve eviction decisions [21]. Last reuse distance, another recency metric, was used by Jiang and Zhang to avert thrashing [17], by Chen et al. to improve Linux VM [8], and by Zhou et al. to improve multi-programming [27]. All of these techniques try to best allocate physical memory for a fixed subset of the working set, but are of limited benefit when the total working set fits in available memory or when the available memory is too small for the subset. Resource-based memory management, on the other hand, can improve performance when given additional physical memory by reducing the frequency of GC. When the memory is in short supply, it increases the frequency of GC to reduce the size of the heap. In this paper we have presented a set of techniques that adapt between smaller heap sizes (more frequent garbage collections) and larger heap sizes (an increased chance of paging) dynamically in a shared environment.

7 Summary

Two major shifts in memory system design are the separation of data and their storage by the virtual memory system and the decoupling of data demand and memory allocation by garbage collection. Resource-based garbage collection connects the program-level allocation and the physical storage by adapting the memory usage with the available resource. The time-memory curve and adaptive GC are effective techniques in avoiding conflicts yet making full uses resources in a dynamically shared environment for garbage-collected programs. We then presented Poor Richard’s Memory Manager, a system which can be added to any collector to improves its paging performance. We demonstrate the effectiveness of Poor Richard’s Memory Manager by using it to improve the paging performance of a range of collectors and systems by an average factor of up to 3.

Acknowledgments

We are grateful to IBM Research for making the Jikes RVM system available under open source terms. The MMTk memory management toolkit was particularly helpful.
References


