OPERATING SYSTEM SUPPORT FOR MODERN APPLICATIONS

A Dissertation Presented

by

TING YANG

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2009

Department of Computer Science

OPERATING SYSTEM SUPPORT FOR MODERN APPLICATIONS

A Dissertation Presented

by

TING YANG

Approved as to style and content by:

____

Emery D. Berger, Co-chair

____

J. Eliot B. Moss, Co-chair

____

Scott F. Kaplan, Member

____

Israel Koren, Member

____

Andrew G. Barto, Department Chair
Department of Computer Science

ABSTRACT

OPERATING SYSTEM SUPPORT FOR MODERN APPLICATIONS

MAY 2009

TING YANG

B.Sc., ZHEJIANG UNIVERSITY, HANGZHOU, CHINA

M.Sc., ZHEJIANG UNIVERSITY, HANGZHOU, CHINA

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Emery D. Berger and Professor J. Eliot B. Moss

Computer systems now run drastically different workloads than they did two decades ago. The enormous advances in hardware power, such as processor speed, memory and storage capacity, and network bandwidth, enable them to run new kinds as well as a large number of applications simultaneously. Software technologies, such as garbage collection and multi-threading, also reshape applications and their behaviors, introducing more challenges to system resource management.

However, existing general-purpose operating systems do not provide adequate support for these modern applications. These operating systems were designed over two decades ago, when garbage-collected applications were not prevalent and users interacted with systems using consoles and command lines, rather than graphical user interfaces. As a result, they fail to allow necessary coordinations among resource management components to ensure consistent performance guarantees. For example, garbage-collected applications cannot adjust themselves to maintain high throughput under dynamic memory pressure, simply because existing virtual memory managers do not collect and expose enough information to them. Furthermore, despite the increasing demand of supporting co-existing interactive applications in desktop environment, resource managers (especially memory and disk I/O) mostly focus on optimizing throughput. They each work independently, ignoring the response time requirements that the CPU scheduler attempts to satisfy. Consequently, pressure on any of these resources can significantly degrade application responsiveness.

In order to deliver robust performance to these modern applications, an operating system has to coordinate its resource managers (e.g., CPU, memory, and disk I/O), as well as cooperate with resource managers in the user space, such as the garbage collector and the thread manger. To support garbage-collected applications, we present CRAMM, a system that enables them to predict an appropriate heap size using information supplied by the underlying operating system, allowing them to maintain high throughput in the face of changing memory pressure. To support highly interactive workloads, we present Redline, a system that manages CPU, memory, and disk I/O in an integrated manner. It uses lightweight specifications to drive CPU scheduling and to coordinate memory and disk I/O management to serve the needs of interactive applications. Such coordination enables it to maintain responsiveness in the face of extreme resource contention, without sacrificing resource utilization. We also show that Redline can be used to support response time sensitive multi-threaded server applications.

Our experiences and extensive experiments show that we can coordinate resource managers, both inside and outside the operating system, efficiently without destroying the modularity of the existing system. Such coordination prevents resource managers from working at cross purposes, and dramatically improve the performance of applications when facing heavy resource contention, sometimes by orders of magnitude.

TABLE OF CONTENTS

Page

CHAPTER

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Motivation

The tremendous advances in computer technology over the past two decades have reshaped the way applications are developed, how they behave, and how users interact with computer systems. These have led to dramatic changes in the workloads running on computer systems, which, in turn, introduce new challenges in operating system resource management, as well as the need to support new kinds of applications.

The exponential increases in hardware power, such as processor speed, memory, storage capacity, and network bandwidth, have enabled computers to run new kinds of applications, as well as new combinations of applications. Inexpensive machines can now run many latency-sensitive and resource-intensive applications (e.g., multimedia players, image processors, web browsers, and video games) that were previously infeasible. For example, a personal computer can simultaneously play a high-quality movie, download and display web pages, and scan for viruses. Furthermore, graphical user interfaces have become the primary means to interact with and manage computers, since they make systems much more user-friendly. Compared to command-line interactive applications, that need a very limited amount of resources and can tolerate perception delays of over 100 ms, modern interactive applications are much more challenging. Multimedia and GUI applications usually need substantially more resources and require perception delays of less than 50 ms to provide users with smooth experiences.

Similarly, new software technologies also have impacts. Garbage collection provides automatic memory management in user space, allowing programmers to develop applications in a faster and safer fashion. Multi-threading enables applications to perform many tasks simultaneously, allowing more efficient use of resources. Unlike traditional applications, new applications have the ability to control the amount of resources they consume, such as the heap size used by garbage-collected applications and the number of threads used by multi-threaded applications. This extra level of management in application runtime systems makes resource management more complicated, but also brings new opportunities, since applications can adapt to resource allocation changes, achieving more reliable performance.

However, existing general-purpose operating systems (e.g., Windows, Linux, FreeBSD, and Solaris) have not kept up with these changes. These operating systems evolved from systems dating from the 1970s and 1980s, when batch and command-line applications dominated and supporting interactivity was relatively easy. These operating systems typically use heuristics to better support the need of modern applications better, such as using priority boosting to achieve better interactiveness. These approaches tend to be ad hoc and inefficient, especially under high resource contention. In general, their resource managers are designed to work independently to optimize overall system utilization. Due to the lack of coordination, resource managers may work at cross purposes or make conflicting decisions when facing resource contention, leading to dramatic performance losses that are avoidable. Here are a few examples.

1.1.1 Problem One: Garbage-collected Applications

The performance of garbage-collected applications is highly sensitive to heap size. A smaller heap reduces the amount of memory referenced, but requires more frequent garbage collections that hurt performance. A larger heap reduces the frequency of collections, thus improving performance by up to 10x. However, if the heap cannot fit in available RAM, performance drops off suddenly and sharply.

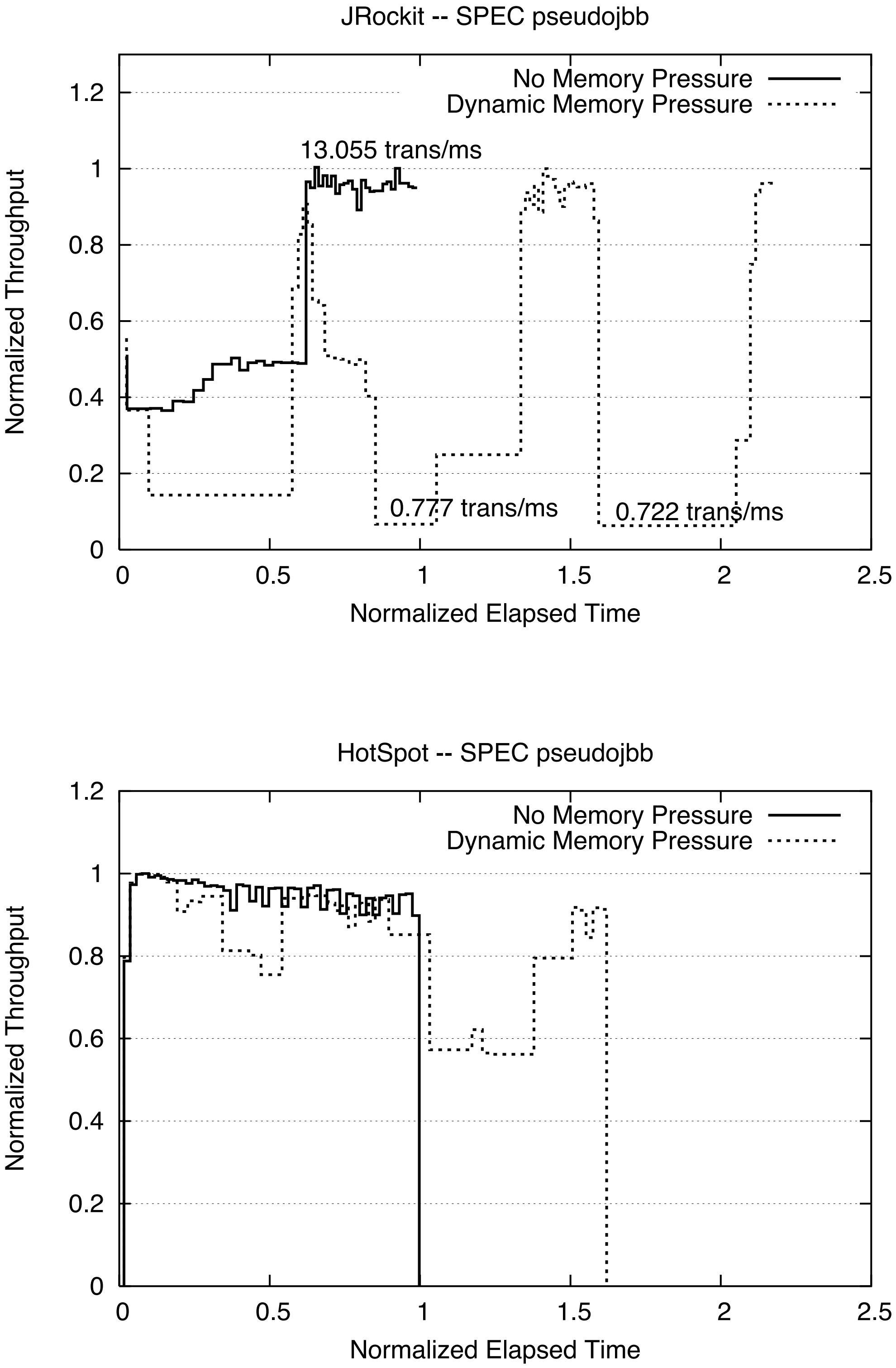


Figure 1.1. Impact of bursts of memory pressure on the performance of two industrial Java virtual machines: JRockit and HotSpot. Page swapping degrades their performance by up to 94% and 40% respectively, and dramatically increases total execution time.

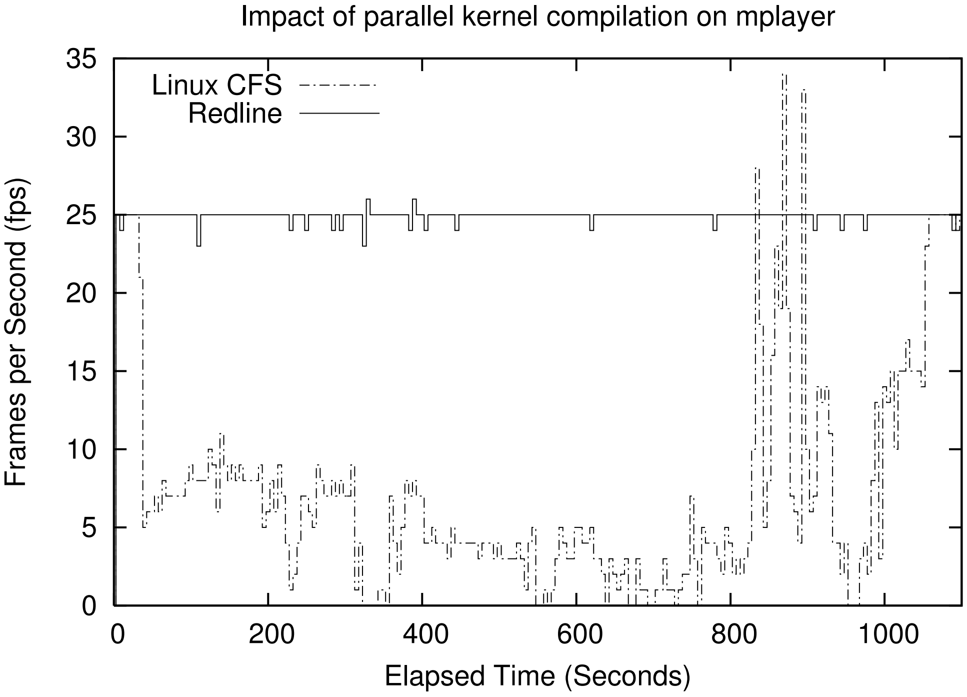


Figure 1.2. The frame rate of mplayer when performing a Linux kernel compiling using make -j32 on a standard Linux 2.6 kernel. Heavy resource contention makes the movie unwatchable.

Due to the fact that the virtual memory manager in the operating system does not expose enough information, existing garbage-collected languages either ignore this problem, allowing only static heap sizes, or adapt the heap size dynamically using mechanisms that are only moderately effective. For example, Figure 1.1 shows the effect of dynamic memory pressure on two industrial-strength Java virtual machines, BEA's JRockit [19] and Sun HotSpot [69], running a variant of the SPECjbb2000 benchmark. The solid line depicts program execution when given enough RAM, while the dashed line shows execution under extra periodic bursts of memory pressure. Due to page swapping, this memory pressure degrades performance by up to 94% (JRockit) and 40% (HotSpot), and dilates overall execution time by a factor of 220% and 160% respectively.

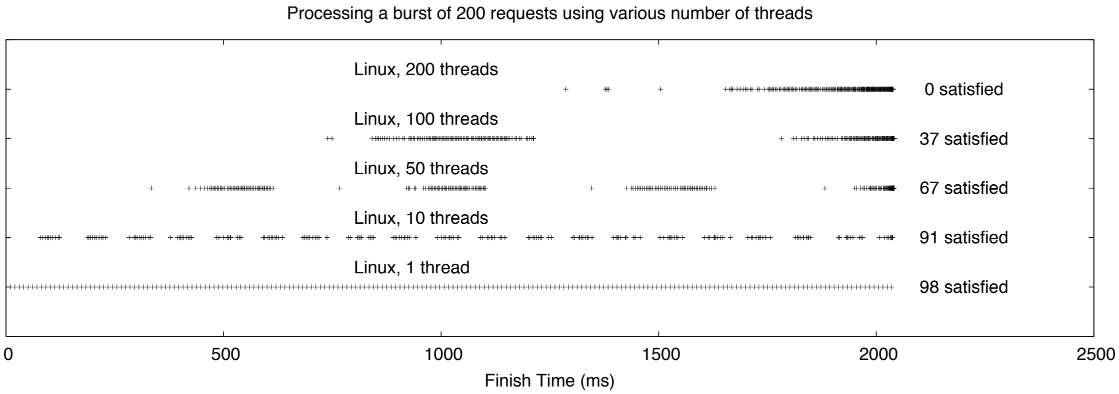


Figure 1.3. Serving a burst of 200 requests using various number of threads under Linux. Each request has about 10 ms computation work and their response time requirement is 1000 ms. Using too many threads causes all requests to miss their deadline.

1.1.2 Problem Two: Interactive Applications

Applications running in desktop environments have become increasingly interactive and resource-intensive, and users rely heavily on graphical user interfaces (GUI) to interact with and manage their computers. The resource managers in current operating systems work independently to optimize overall system utilization. They use general policies that do not take response time requirements into account, providing at best limited support for interactive applications.

Take Linux as an example. Figure 1.2 shows the frame rate of mplayer, a movie player, while a Linux kernel compilation, invoked using make -j32, is performed using both a standard Linux 2.6.x kernel and our Redline kernel. The thirty-two parallel kernel compilation jobs soon exhaust system resources, causing severe contention for the CPU, memory, and disk I/O. This workload substantially degrades the interactivity of the movie player, rendering the video unwatchable. Worse, the entire GUI becomes unresponsive. Similar behavior occurs on Windows when watching a video using the Windows Media Player while running a virus scanner in the background.

1.1.3 Problem Three: Multi-threaded Server Applications

Many server applications are now multi-threaded, which allows them to make better use of hardware resources, such as exploiting multiple CPUs and hiding disk latencies. Choosing a proper number of threads is challenging and important to performance. If the number of threads is too small, a server application may lose the opportunity to overlap I/O and network latencies with computation, resulting in poor throughput. Using more threads helps to improve resource utilization, but too many threads can significantly compromise server response times in the face of a large burst of requests.

Figure 1.3 shows how a burst of 200 requests are served under Linux using various number of threads. Each request consists of about 10 ms of computation work, and the desired response time is 1000 ms. The graph shows when each request is finished and the number of requests satisfied within the desired time for each case. As the number of threads increases, each thread receives less CPU bandwidth, requiring more time to process a request. As a result, the number of requests that meet the desired response time drops. Eventually, all requests miss their deadline when 200 threads are used.

1.1.4 Why Existing Operating Systems are Insufficient

The reason why existing operating systems fail to provide adequate support for modern applications lies in their resource management. Although these systems have improved significantly over the years, their general principles have remained almost unchanged. We will provide more details when we discuss specific problems in the later chapters of this dissertation.

1.1.4.1 CPU Management

Designed to support batch workloads, the primary concern of traditional CPU schedulers is to allow applications to share the CPU bandwidth fairly.

A time-sharing scheduler, as in FreeBSD [91], Solaris [90] and Windows [112], allocates CPU quanta (the unit for CPU allocation) to applications based on their dynamic priorities. Staring from an initial value, the scheduler periodically re-evaluates each application's priority based on its CPU usage. The priority of an application is decreased after consuming CPU and increased after waiting or blocking. The scheduler selects the application with the highest priority and assigns the next quantum to it.

The weight-based proportional-share scheduler used by Linux [27] allocates CPU to applications, so that they make progress proportional to their weights. Higher weights indicate more CPU bandwidth. The scheduler normalizes the progress made by each application with respect to the total weight in the system, and selects the application that has made the least progress to be executed next. For both of these schedulers, CPU bandwidth allocation is relative, i.e., the more applications that are running, the less CPU bandwidth each of them receives.

These schedulers neither take response time constraints into account nor protect against overload. As a result, they lack the ability to support diverse workloads including GUI and multimedia applications. They cannot provide isolation between threads and applications, and consequently they cannot provide meaningful guarantees about the level of service that applications will receive.

1.1.4.2 Memory Management

For an application to run smoothly, it is essential to cache enough information in memory to avoid expensive disk I/O operations. In order to do so, most operating system uses an application's *working set* to predict the information that it likely to use in the near future, which is defined by Denning [49] as:

> *The working set $W(t,\tau)$ of a process at time t is the collection of information referenced by the process during the process time interval $(t-\tau,t)$.*

The virtual memory manager (VMM) identifies the working set by scanning memory pages periodically. Those pages that have not been referenced since the last scan are considered out of working set and subject to eviction. Linux [27] uses a global page list that approximates LRU order, while Windows [112] organizes pages on a per-process basis. The scan of memory pages is triggered when the VMM fails to satisfy an allocation request.

As memory pressure increases, the VMM will scan memory pages more frequently (i.e., using a smaller τ), shrinking the working sets of applications to fit all of them in. This works reasonably well when memory pressure is moderate, since it optimizes the total number of potential page faults and offers graceful degradation. However, if memory pressure is high enough, it leads to a well-known phenomenon–*thrashing*, where the working sets cached in memory are so small that no application can make reasonable progress. Furthermore, such a page reclamation policy favors applications that have higher memory reference speeds, regardless of CPU scheduling priorities.

1.1.4.3 Disk I/O Management

Accessing disk is expensive, and thus has a significant impact on system throughput. The amount of time needed to serve a disk I/O request consists of three parts: *transfer time*, *seek time*, and *rotation time*. The latter two mechanical delays are much longer and dominate the overhead of disk I/O operations. Simply serving requests in FIFO order leads to prohibitively low throughput. Therefore, a system usually groups I/O requests and serves them in a certain order to reduce mechanical delays. The SCAN algorithm used by Linux [27], FreeBSD [91], and Solaris [90] is a typical example. SCAN sorts requests according to their physical block locations. It serves requests in the sorted order regardless of which applications they belong to, favoring overall throughput over the timeliness of individual requests. In addition, most systems also put the data read from and written to disk in a memory cache to avoid unnecessary I/O operations. Technically speaking, this cache belongs to memory management. Dirty data in this cache is flushed to disk periodically en masse to achieve better throughput.

To summarize, the resource managers neither cooperate with each other, nor communicate with user applications. They make decisions based on their own general policies, attempting to maximize the overall system throughput without exposing much information to user applications. They may work at cross purposes, and thus decisions made by different resource managers can interfere with each other. While such a scheme works reasonably well for supporting batch workloads, it lacks the ability to support the diverse needs of modern applications, such as providing meaningful response time guarantees for interactive applications, and supplying garbage-collected applications with the necessary information for choosing their heap sizes.

1.2 The Thesis

The thesis that this dissertation supports is:

A general-purpose operating system needs to and can be extended to serve the needs of modern applications better by coordinating its resource managers, as well as cooperating with user-level resource management.

The non-cooperative resource management scheme used by existing general-purpose operating systems can lead to a wide range of problems, especially when facing resource contention and when resource managers make conflicting decisions. This dissertation focuses on three such problems with a significant impact on current systems:

- *choosing the heap size for garbage-collected applications*, which involves cooperation between the VMM and the garbage collector;
- *maintaining responsiveness for interactive applications*, which requires coordinating multiple resources, including the CPU, memory, and disk I/O; and
- *maximizing the responsiveness of multi-threaded server applications*, which requires cooperation between the CPU scheduler and the thread manager.

1.2.1 System Design Requirements

In order to solve these problems, a general-purpose operating system has to be extended with mechanisms that enable the necessary communication and cooperation among resource managers. The design of these extensions must consider a number of requirements to ensure that the system is both effective and practical.

- *Modularity:* The interface between different components should be minimal, so that the modularity of these components can be preserved and they remain manageable individually.

- *Low overhead:* The extensions made should have well-controlled overhead, so that the system has comparable performance to general purpose operating systems while adding the ability to coordinate resource managers.

- *Compatibility:* The system should have a certain degree of backward compatibility, so that legacy applications can take advantage of new features without any modification, or with minimal modification if required.

- *Ease of use:* The system should not impose too much of a burden on users. It should allow users to use any new features with minimal effort and manage them intuitively, without deep knowledge of how resources are managed.

1.3 Contributions

This dissertation explores the possibilities of solving the problems described above using *integrated resource management*, which includes both coordination of kernel resource managers themselves and cooperation between kernel and user-level resource mangers. Our approaches comprise the following contributions.

1.3.1 CRAMM for Garbage-collected Applications

For garbage-collected applications, we present **CRAMM** (Cooperative Robust Automatic Memory Management), a system that enables cooperation between the *virtual memory manager* in the operating system and the *garbage collector* in the virtual machine. It allows garbage-collected applications to adjust their heap sizes dynamically in response to changing memory allocation, and thus achieve high throughput.

On the virtual memory manager side, CRAMM uses a novel low overhead mechanism to track application working set size. Upon request, the virtual memory manager reports to the garbage collector the working set size together with the amount of available memory in the system. The garbage collector then feeds this information to an analytical heap sizing model, selecting a new heap size whose corresponding working set size just fits in the amount of available memory. This allows garbage-collected applications always to use nearly optimal heap sizes, maximizing throughput while incurring at most a trivial amount of page swapping.

1.3.2 Redline for Interactive Environments

For highly-interactive environments, we present **Redline**, a system that integrates resource management (memory and disk I/O management) with the CPU scheduler, orchestrating these resource managers to maximize the responsiveness of interactive applications.

Redline adopts a lightweight *specification*-based approach that provides enough information to allow the system to meet the response time requirements of interactive applications. Redline's specifications, which essentially are *CPU reservations* [74], give a rough estimate of the amount of resources required by an application over any period of time in which they are active. These specifications allow applications to reserve CPU bandwidth, and are used to coordinate other resource managers with the CPU scheduler. They are concise, consisting of just a few parameters, and are straightforward to generate.

Each resource manager then uses these specifications to inform its decisions. Redline's memory manager protects the working sets of interactive applications according to their specifications, preferentially evicts pages from best-effort applications, and further reduces the risk of paging through a rate-controlled memory reserve. Redline's disk I/O management avoids pauses in interactive applications by dynamically prioritizing these tasks based on their specifications. Finally, Redline extends a standard time-sharing CPU scheduler with an earliest deadline first (EDF)-based scheduler [85] that implements the *CPU reservation* mechanism to serve the needs of interactive applications according to their specifications.

1.3.3 Redline for Multi-threaded Server Applications

For multi-threaded server application, we present a Redline server architecture that cooperates with the underlying Redline EDF scheduler to achieve better response time guarantees. The Redline server uses recent workload information provided by the CPU scheduler to determine the maximum number of requests it can process simultaneously. It attaches to each of these requests a specification generated from the desired response time, and passes them to interactive threads. The Redline EDF scheduler then serves them using the CPU reservation according to the attached specifications. Through careful management of the specifications, the Redline server is thus able to maximize the number of requests satisfied without sacrificing throughput.

1.4 Outline of the Dissertation

We organize this dissertation into three parts. The first part describes the CRAMM system, starting by presenting its motivation and an overview in Chapter 2. Chapter 3 proposes the heap sizing model based on a comprehensive analysis, and Chapter 4 describes how the CRAMM VM supports dynamic heap sizing. Chapter 5 provides an extensive evaluation of our prototype implementation in Linux.

The second part focuses on the Redline system. Chapter 6 provides additional motivations for Redline, as well as an overview. Chapter 7 describes its specification management. The design of Redline's CPU, memory, and disk I/O management are described in Chapters 8, 9 and, 10 respectively. Chapter 11 evaluates the effectiveness of the Redline system in desktop environments by stressing the system with extreme workloads.

The third part (Chapter 12) discusses how Redline can also be used to support multi-threaded server applications. It presents the Redline server architecture and a proof-of-concept evaluation. Finally, Chapter 13 compares both CRAMM and Redline with related work. Finally, we present conclusions and suggestions for future work in Chapter 14.

CHAPTER 2

CRAMM FOR GARBAGE-COLLECTED APPLICATIONS

Garbage-collected languages have now become mainstream, including general-purpose languages such as Java and C# and scripting languages such as Python and Ruby. Garbage collection provides many software engineering advantages over traditional manual memory management, eliminating dangling pointers and double freeing, and drastically reducing the risk of memory leaks. However, it also carries a potential liability: *page swapping*, a problem known for decades. It has been observed that when the heap accessed by garbage collection is larger than available physical memory, the collector spends most of its time *thrashing* [98]. Because disks are five to six orders of magnitude slower than RAM, page swapping can easily ruin performance.

In this part of the dissertation, we study the behavior of garbage-collected applications and explore how the virtual memory manager in the operating system can provide support so that GCed applications can choose their heap size in a predictive manner, making full use of memory while avoiding page swapping.

2.1 Garbage Collection Heap Size

The virtual memory (VM) systems in today's operating systems were designed to support applications written in the widely-used programming languages of the 1980s and 1990s, C and C++. To maximize the overall performance of these applications, it is enough to fit their working sets in physical memory [50]. Since these applications do not have the ability to change their working set sizes, existing VM systems typically manage available memory with an approximation of LRU [33, 40, 49, 50, 89], which works reasonably well for legacy applications.

Garbage collection offers an extra layer of memory management in user space. It can run applications with different heap sizes, effectively changing their working set sizes. Therefore, garbage-collected application performance is highly sensitive to heap size. A smaller heap reduces the amount of memory referenced, but requires more frequent garbage collections that may hurt performance. A larger heap reduces the frequency of collections, thus improving performance by up to 10x. However, if the heap cannot fit in available RAM, performance drops off suddenly and sharply. This is because garbage collection has a large working set (a full collection visits almost the entire heap) and thus can trigger catastrophic page swapping that degrades performance and increases collection pauses by orders of magnitude [67]. Hence, heap size and main memory allocation need to be coordinated to achieve good performance. Unfortunately, current VM systems do not provide sufficient support for this coordination, and thus do not support garbage-collected applications well.

Choosing the appropriate heap size for a garbage-collected application—one that is large enough to maximize throughput but small enough to avoid paging—is a key performance challenge. The ideal heap size is one that makes the working set of garbage collection just fit within the process's main memory allocation. However, an *a priori* best choice is impossible in multiprogrammed environments where the amount of main memory allocated to each process constantly changes. Existing garbage-collected languages either ignore this problem, allowing only static heap sizes, or adapt the heap size dynamically using mechanisms that are only moderately effective, as we have shown in Chapter 1.

The problem with these adaptive approaches is not that their adaptivity mechanism is broken, but rather that they are *reactive*. The only way these systems can detect whether the heap size is too large is to wait until paging occurs, which leads to unacceptable performance degradation. Since they lack adequate information, these systems can reduce the heap size only incrementally, prolonging the painful period of dynamic memory pressure.

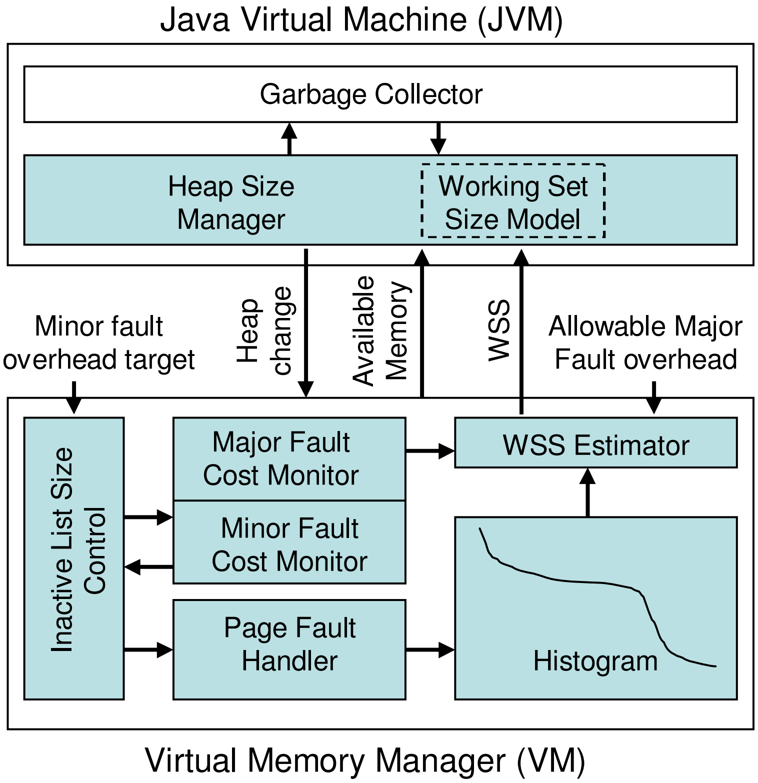


Figure 2.1. The CRAMM system. The CRAMM VM system efficiently gathers detailed per-process reference information, allowing the CRAMM heap size model to choose an optimal heap size dynamically.

2.2 An overview of the CRAMM system

CRAMM (Cooperative Robust Automatic Memory Management) is a system that enables garbage-collected applications to *predict* an appropriate heap size, allowing the system to maintain high performance while adjusting dynamically to changing memory pressure. Figure 2.1 presents an overview of the CRAMM system.

CRAMM consists of two parts. The first part is the CRAMM VM system that dynamically gathers the *working set size (WSS)* of each process, where we define the WSS as *the main memory allocation that yields a trivial amount of page swapping*. To accomplish this, the VM system maintains separate page lists for each process and computes an *LRU reference histogram* [116, 133] that captures detailed reference information while incurring little overhead (around 1%). The second part of CRAMM is its heap sizing model, which controls application heap size and is independent of any particular garbage collection algorithm. The CRAMM model compares the WSS measured by the CRAMM VM to the current heap size. It then uses this comparison to select a new heap size that is as large as possible (thus maximizing throughput) while yielding little or no page faulting behavior. We apply the CRAMM model to five different garbage collection algorithms, demonstrating its generality.

CHAPTER 3

CRAMM: GARBAGE COLLECTION HEAP SIZING MODEL

A *garbage collector (GC)* periodically and automatically finds and reclaims heap-allocated objects that a program can no longer possibly use, during which it may reference a large amount of memory pages within a short period of time. In order to maintain high performance, it must have the ability to predict the working set size corresponding to a choosen heap size.

In this chapter, we first present a general discription of garbage collection, introducing GC terminology and concepts that are critical to understanding CRAMM. We then analyze the behavior of garbage collection, revealing how heap size impacts working set size and total execution time. Based the results of our analysis, we present CRAMM's heap sizing model, and describe its adaptive adjustment algorithm.

3.1 Garbage Collection Background

Garbage collectors operate on the principle that if an object is *unreachable* via any chain of pointers starting from *roots*—pointers found in global/static variables and on thread stacks—then the program cannot possibly use the object in the future, and the collector can reclaim and reuse the object's space. Through a slight abuse of terminology, reachable objects are often called *live* and unreachable ones *dead*. Reference counting collectors determine (conservatively) that an object is unreachable when there are no longer any pointers to it. Here, we focus primarily on *tracing collectors*, which actually trace through pointer chains from roots, visiting reachable objects.

The frequency of collection is indirectly determined by the *heap size*: the maximum virtual memory space that may be consumed by heap-allocated objects. When allocations have consumed more than some portion of the heap size (determined by the collection algorithm), collection is invoked. Thus, the smaller the heap size, the more frequently GC occurs, and the more CPU time is spent on collection.

3.1.1 Heap Orgnization

GC algorithms divide the heap into one or more *regions*. A *non-generational* GC collects all regions during every collection, triggering collection when some percentage of the entire heap space is filled with allocated objects. A non-generational GC may have only one region. In contrast, *generational* GCs partition the regions into groups, where each group of regions, called a *generation*, contains objects of a similar age. Most commonly, each group consists of a single region. When some percentage of the space set aside for a generation has been filled, that generation, and all younger ones, are collected. Additionally, live objects that survive the collection are generally *promoted* to the next older generation. New objects are typically allocated into a *nursery* region. This region is usually small, and thus is collected frequently, but quickly (because it is small). The generational configurations that we consider here have two generations, a nursery and a *mature space*. Because nursery collection generally filters out a large volume of objects that die young, mature space grows more slowly—but when it fills, that triggers a *full heap* collection.

3.1.2 Reclaiming Dead Objects

Orthogonal to whether a collector is generational is how it reclaims space. *Mark-sweep (MS)* collection marks the reachable objects, and then sweeps across the allocation region to reclaim the unmarked ones. MS collection is *non-copying* in that it does not move allocated objects. In contrast, a *copying* collector proceeds by copying reachable objects to an empty copy space, updating pointers to refer to the new copies. When done, it reclaims the previous copy space. A typical example of copying collector is *Semi-space (SS)*, which decides the heap into to equal spaces and copies live objects from one space to the other. We do not consider here collectors that compact in place rather than copying to a new region, but our techniques would work just as well for them. Their paging performance is similar to MS collectors. Notice that collectors that have a number of regions may handle each region differently. For example, a given GC may collect one region by copying, another by MS, and others it may never collect (so-called *immortal spaces*).

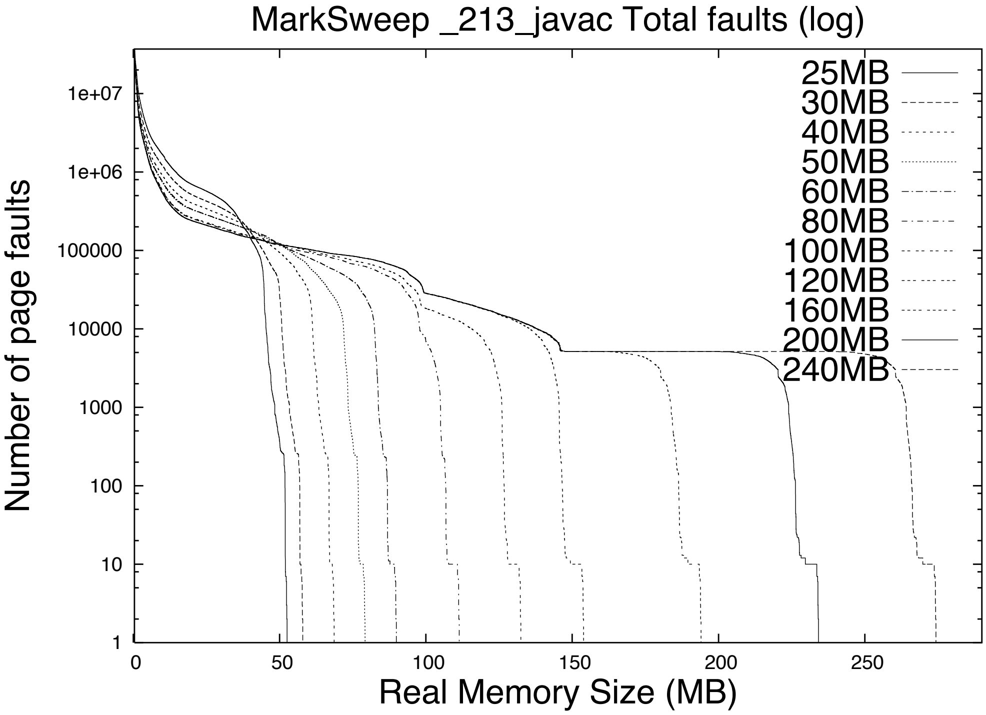
During the execution of a garbage-collected application, allocation and collection are intertwined. When allocating into an MS-managed region, the allocator uses free lists to find available chunks of space. When allocating into a copying region, it simply increments a free space pointer through the initially empty space. For generational collection, the nursery is usually a copy-collected space, thus allowing fast allocation. The mature space, however, may be a copying- or a non-copying-collected region, depending on the particular collector.

3.2 Garbage Collection Behavior Analysis

To build robust mechanisms for controlling paging behavior of GCed applications it is important first to understand their behaviors. We studied them by analyzing memory reference traces for a set of benchmarks, executed under each of several collectors, for a number of heap sizes (our earlier work explores this in more detail [135]). The goal was to reveal, for each collector, the regularities in the reference patterns and the relation between heap size and working set size.

3.2.1 Garbage Collection Paging Behavier

Figures 3.1 shows the number of page faults for varying physical memory allocations, for SPECjvm98 benchmark `javac` under two different collectors: mark-sweep (MS) and semi-space (SS). Each curve comes from one simulation of the benchmark in question, at a particular fixed heap size. (Note that the vertical scales are *logarithmic*.)



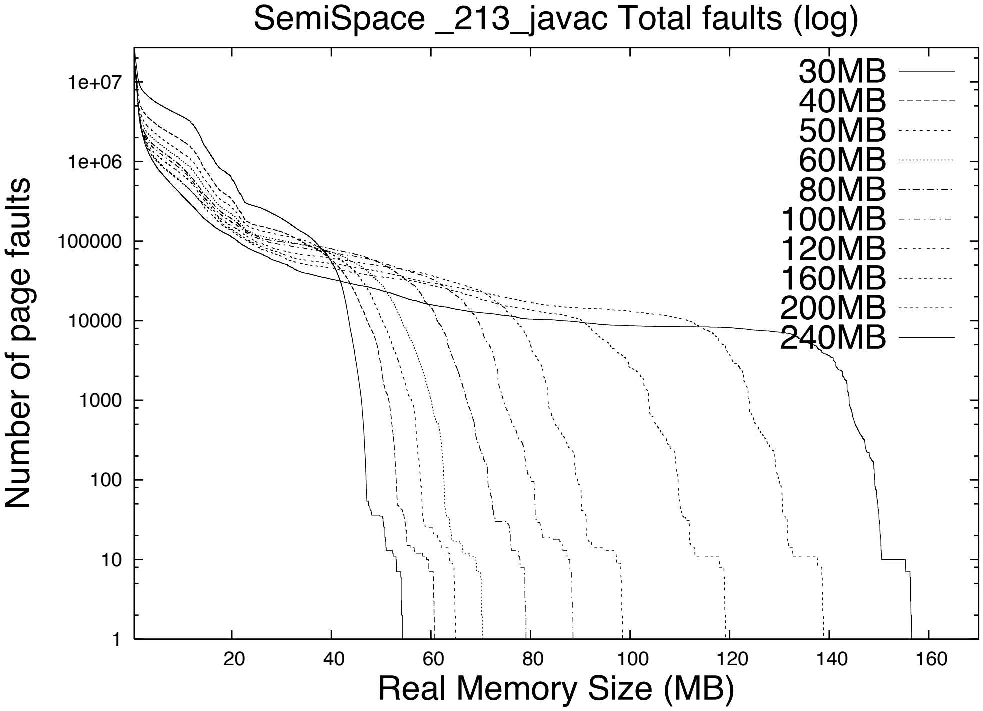


Figure 3.1. The number of page faults encountered when varying physical memory allocation for a set of heap sizes. (Garbage Collector: MarkSweep and SemiSpace, Benchmark: `_213_javac`)

The behaviors of these collectors strongly resemble each other. We see that each curve has three regions. At the smallest real memory sizes, we see extremely high paging. Curiously, larger *heap* sizes perform better for these small *real memory* sizes! This happens because most of the paging occurs during collection, and a larger heap size causes fewer collections, and thus less paging.

The second region of each curve is a broad flat area representing substantial paging. For a range of physical memory allocations, the program repeatedly allocates in the heap until the heap is full, and the collector then walks over most of the heap, collecting dead objects. Both steps are similar to looping over a large array, and require an allocation equal to a semi-space (for SS) or the whole heap (for MS) to avoid paging.

Finally, the third region of each curve is a sharp drop in faults that occurs once the allocation is large enough to capture the "looping" behavior. Note that the final drop in each curve happens in order of increasing heap size, i.e., the smallest heap size drops to zero page faults at the smallest allocation. It occurs at an allocation that is a constant plus nearly half of the heap size for SS, or a constant plus the heap size for MS. This regularity suggests that there is a base amount of memory needed for the Java virtual machine and application code.

From this analysis we see that the working set size (WSS) for a GCed application is determined almost entirely by what happens during full collections, because full collections touch every reachable heap object. Since live and dead objects are generally mixed together, the working set includes all heap pages used for allocated objects. It also includes the space needed for copied survivors of copying regions. Thus, each non-copying region contributes its size to the working set, while each copying region adds its size *plus* the volume of copied survivors, which can be as much as the size of the copying region in the worst case.

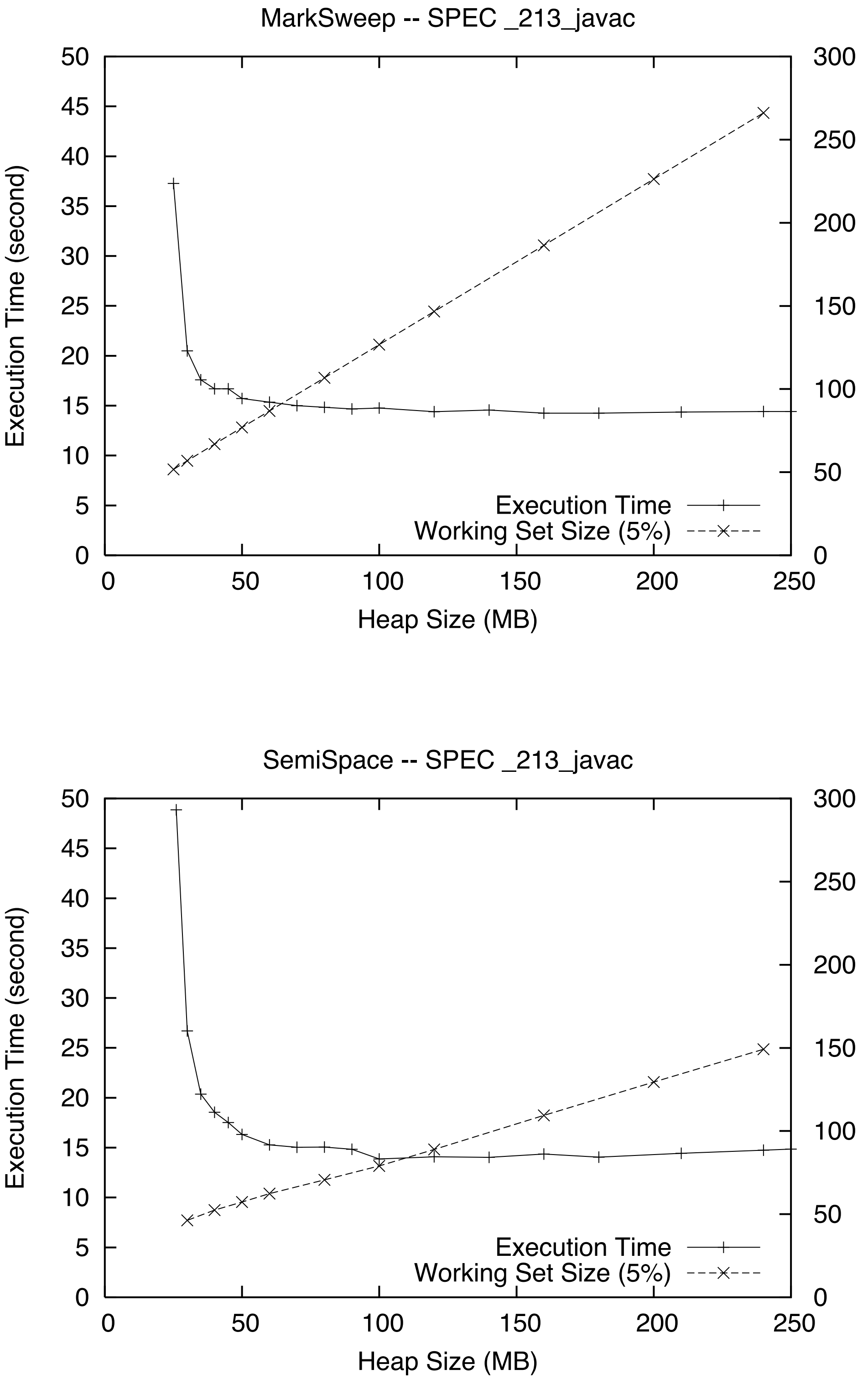


Figure 3.2. The effect of heap size on performance and working set size (the number of pages needed to run with 5% slowdown from paging).

3.2.2 Heap Size, Execution Time and Working Set Size

Several properties of GCed applications are important here. First, given adequate physical memory, performance varies with heap size. For example, Figure 3.2 depicts the effect of different heap sizes on performance. Similar to the previous one, this graph is for `javac` under two garbage collectors (SS and MS), but it is typical of other benchmarks. On the left-hand side, where the heap is barely large enough to fit the application, execution time is high. As the heap size increases, execution time sharply drops, finally running almost 2.5x faster. This speedup occurs because a larger heap reduces the number of collections, thus reducing GC overhead. The execution time graph has a $1/x$ shape, with vertical and horizontal asymptotes.

However, the *working set size*—here given as the amount of memory required to run with at most 5% elapsed time added for paging—has a linear shape. The slope for MS is approximately 1.0, since it always fills up the whole heap before collection. On the other hand, the slope for SS is approximately 0.5, because it alway fills up half of the heap before collection due to its copying nature. Overall, the key observation is that working set size is very nearly linear in terms of heap size, and can be predicted reasonably well.

3.3 GC-neutral Heap Sizing Model

The goal of the CRAMM heap sizing model is to relate *heap size* and *working set size*, so that, given a current physical memory allocation, we can determine a heap size whose working set size just fits in the allocation. We define heap size, H, as the maximum amount of space allowed to contain heap objects (and allocation structures such as free lists) at one time. If non-copy-collected regions use N pages and copy-collected regions allocate objects into C pages, then

$$H = N + 2 \times C$$

Note that we must reserve up to C pages into which to copy survivors from the original C space, and the collector needs both copies until it is done. The total WSS for the heap during full collection is determined by the pages used for copied survivors, *CS*:

$$WSS = N + C + CS$$

Thus heap *WSS* varies from $N + C$ to $N + 2 \times C$.

As a program runs, its usage of non-copying and copying space may vary, but it is reasonable to assume that the balance usually does not change rapidly from one full collection to the next. We call the ratio of allocable space ($N + C$) to heap size ($N + 2 \times C$) the *heap utilization*, u. It varies from 50% for $N = 0$ (i.e. pure copying collectors, such as SS) to 100% for $C = 0$ (i.e. pure non-copying collectors, such as MS). Given an estimate of u, we can determine $N + C$ from H, but to determine *WSS* we also need to estimate *CS*. Fortunately, *CS* is a property of the application (volume of *live* objects in copy-collected regions), not of the heap size. As with u, we can reasonably assume that *CS* does not change too rapidly from one full collection to the next.

When adjusting the heap size, we use this equation as our model:

$$\Delta H = (\Delta WSS - \Delta CS)/u$$

According to this equation, a garbage-collector needs several pieces of information to determine how much the heap size should be changed to fit itself in the available memory. The heap utilization u and ΔCS can be estimated by the garbage collector itself based on recent history. ΔWSS is just the target WSS (i.e., the physical memory allocation the operating system is willing to give to the application) minus the WSS corresponding to the current heap size. The garbage collector will rely the underlying CRAMM virtual memory system to provide both of these facts (i.e., the physical memory allocation to the applications and the WSS corresponding to the current heap size), when adjusting the heap size.

3.4 Adjusting the Heap Size

Now we describe how the JVM can use the model presented in the previous section to adjust the heap size in response to dynamic memory allocation changes. Since the analytical heap sizing model is neutral with respect to garbage collection algorithms, it easy to apply it to different garbage collectors. We first consider how to determine the initial heap size. We then describe the basic adjustment algorithm for the common case, which happens after every *full* collection. Finally, we add an adjustment to handle *n*ursery collections for generational collectors.

3.4.1 Selecting the Initial Heap Size

Once the JVM reaches the point where it needs to calculate an initial heap size, it has touched an initial working set of code and data. Thus, the space available for the heap is exactly the volume of free pages the VM is willing to grant us (call that *Free*). However, at this stage, the JVM does not have any information to drive the heap sizing model, therefore it has to choose an initial heap size conservatively to avoid page swapping during the first full collection. We wish to set our heap size so that our worst case heap WSS during the first full collection will not exceed *Free*. But the worst case heap WSS is exactly the heap size, so we set H to the minimum of *Free* and the user-requested initial heap size.

3.4.2 Base Heap Adjustment Algorithm

After each full heap collection, the garbage collector re-computes the heap size that should be used. Pseudo-code 1 shows the heap size adjustment algorithm. The most important responsibility of this algorithm is tracking the parameters, specifically the heap utilization u and ΔCS. To determine u, we simply calculate it as the amount of memory allocated divided by the heap size (line 3), and assume that the near future will be similar.

Estimating ΔCS is more involved and critical (line 4 – 12), because underestimating CS of the next full collection (and thus ΔCS) can result in substantial page swapping. For a pure non-copying collector, this will not be an issue. In order to provide a safe estimate,

Pseudo-code 1 Heap Size Adjust Algorithm

```
 1: get $available\ memory$ and $WSS$ from CRAMM VM;
 2:
 3: $heapUtil = allocated/heapSize$;              // calculate $heapUtil$ factor
 4: **if** ($CS > maxCS$) {                        // observed an increase in $CS$
 5:   update $maxCSInc$;
 6:   update $maxCS$;
 7:   $WSS$ += $maxCSInc/2$;                      // add half of the increase to next $CS$
 8: }
 9: **else**                                       // observe a decrease in $CS$
10:   $WSS$ += $maxCS - CS$;                      // use $maxCS$ as next $CS$
11: $maxCSInc = maxCSInc \times 0.5$;             // decay $maxCS$ and $maxCSInc$
12: $maxCS = maxCS \times 0.98$;
13:
14: $\Delta heap = (available\ memory\ \text{-}\ WSS)/heapUtil$;
15: **new heapSize** $= heapSize + \Delta heap$;  // calculate the new heap size
```

we track the maximum *value* for *CS* that we have seen so far, *maxCS*, and we also track the maximum *increment* we have seen to *CS*, *maxCSInc*.

If, after a full collection, *CS* exceeds *maxCS*, we assume *CS* is increasing and estimate $\Delta CS = maxCSInc/2$, i.e., that it will grow by 1/2 of the largest *increment*. This prevents us from growing the heap size too much when the application is still building up its data structure. Otherwise we estimate ΔCS as $maxCS - CS$, i.e., that *CS* for the next full collection will equal *maxCS*. This prevents us from growing the heap in response to a collection with exceptionally small survival rate. After calculating ΔCS, we *decay maxCS*, multiplying it by 0.98 (a conservative policy, allowing us to grow the heap gradually if the live size of the application becomes smaller), and *maxCSInc*, multiplying it by 0.5 (a more rapidly adjusting policy, allowing us to grow the heap quickly once the application's live size reaches a stable point).

Once all the parameters are updated, we calculate the new heap size (line 14 – 15) using the information supplied by the CRAMM virtual memory manager, specifically the amount of physical memory available to us and the working set size of the current heap size. We also should be careful not to set the heap size to be smaller than the application's live size, which typically triggers an out-of-memory exception.

3.4.3 Handling Nursery Collections

Because nursery collections for generational collectors do not process the whole heap, their WSS reported from the CRAMM VM dramatically underestimates the working set size for future full collections. Furthermore, their survival rate σ (i.e., CS/ν, where ν is the size of the nursery) could be substantially different from a full collection. Predicting the parameters based on a tiny nursery collection is dangerous. Therefore, if the nursery size is less than 50% of allocable space, we do not update H. For larger nurseries, we compensate WSS by adding the size of uncollected copying space times $1+\sigma$:

$$WSS = WSS + (1+\sigma) \times \mathit{uncollected\ size}$$

The purpose of this adjustment is to mimic the working set size of a full collection that has the same survive rate.

This algorithm takes into account the arbitrary combination of copying and non-copying regions and handles generational collectors. Our tracking of *maxCS* and *maxCSInc* also helps avoid paging when the application's live size fluctuates dramatically. We periodically request the current *Free* value, so that we can reduce the heap size between full collections if our allocation shrinks suddenly. If *Free* is less than *maxCS*, we trigger an immediate collection.

CHAPTER 4

CRAMM: VIRTUAL MEMORY SUPPORT

We now present the CRAMM VM system. Given the heap sizing model presented in Chapter 3, the underlying VM system must provide to a GC-based process both its working set size (WSS) and its physical memory allocation,[1] thus allowing the GC to choose a proper heap size. Unfortunately, we cannot easily obtain this information from standard VM systems, including the Linux VM.

We first describe why standard VM systems are insufficient for predictively adaptive heap sizing. We then describe the structure of the CRAMM VM, followed by detailed discussions of how the VM calculates working set sizes and how it controls its overhead.

4.1 CRAMM VM Design

Linux uses a global page replacement policy that manages each physical page within a single data structure for all processes and files. Linux thus has only *ordinal* information about all pages, giving each page a ranking among the total pool of pages. It has no *cardinal* information about the reference rates, nor any separation of pages according to process or file. Consequently, it cannot track the *LRU reference histogram*—the distribution of memory references to pages managed by an LRU queue—which is needed to determine the WSS for each process. Furthermore, it cannot predict how much it could reduce the allocations of files and other processes without inducing heavy page faulting. It therefore

[1] The *physical memory allocation* is not the same as the *resident set size*. The latter is the amount of physical memory currently consumed by a process, while the former is the amount of physical memory that the VM is willing to let the process consume before evicting its pages.

cannot wisely choose a physical memory allocation to offer to a GC-based process. Finally, even if it chose to reduce the allocations for some files or other processes, global page replacement cannot guarantee that it will replace the pages of those processes first.

4.1.1 Collecting Information

The CRAMM VM system addresses these limitations. Figure 2.1 gives an overview of the CRAMM VM structure and interface. For each file and process, the VM keeps separate page lists and an LRU reference histogram. It also tracks the mean cost of a major page fault (one that requires disk I/O) so that, along with the histogram and a desired maximum fault rate, it can compute the WSS of a process.

Its ability to compute the WSS of each file and process allows the CRAMM VM to calculate new allocations to each without causing thrashing by assigning too small an allocation. When an allocation is reduced, the separate page lists allow the VM to prefer reclaiming pages from those files and processes that are consuming more than their allocation.

4.1.2 Cooperating with Garbage Collector

A garbage collector communicates with the CRAMM VM through system calls. First, the collector registers itself as a cooperative process with the CRAMM VM at initialization time. The VM responds with the current amount of free memory, allowing the collector to pick a reasonable initial heap size. Second, after each heap collection, the collector requests a WSS estimate and a physical memory allocation from the VM. The collector then uses this information to select a new heap size. If it changes its heap size, it calls on the VM to clear its old histogram, since the new heap size will exhibit a substantially different reference pattern.

Finally, the collector periodically polls the VM for an estimate of the *free memory*—the physical memory space that could be allocated to the process without causing others to thrash. If this value is unexpectedly low, then memory pressure has suddenly increased.

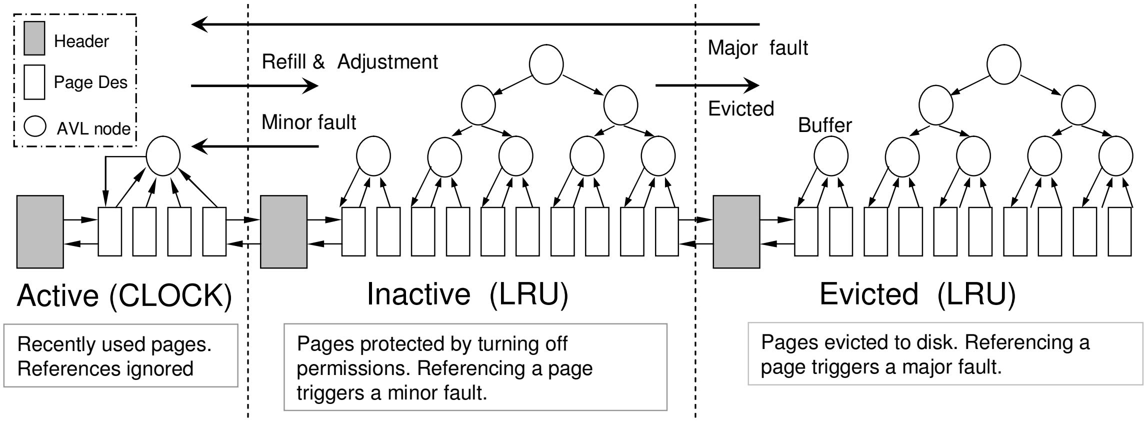

Header
Page Des
AVL node
Refill & Adjustment
Minor fault
Major fault
Evicted
Buffer
Active (CLOCK)
Inactive (LRU)
Evicted (LRU)
Recently used pages. References ignored
Pages protected by turning off permissions. Referencing a page triggers a minor fault.
Pages evicted to disk. Referencing a page triggers a major fault.

Figure 4.1. Segmented queue page lists for one address space (file or process).

Either some other system activity is aggressively consuming memory (e.g., the startup of a new process), or this process has more live data (increased *heap utilization*), and thus is using more memory than expected. The collector responds by pre-emptively collecting the heap and selecting a new heap size.

4.2 CRAMM VM Structure

The CRAMM VM allocates a data structure, called a `mem_info`, for each *address space* (an `inode` for files or an `mm_struct` for processes). This structure comprises a list of pages, an LRU reference histogram, and some additional control fields.

Figure 4.1 shows the page list structure of a process. The CRAMM VM manages each *address space* (the space of a file or a process) much like the Linux VM manages its global queue. For the in-memory pages of each address space, it maintains a *segmented queue* (SEGQ) structure [9], where the *active list* contains the more recently used pages and the *inactive list* contains those less recently used. When a new page is faulted into memory, the VM places it at the head of the active list. If the addition of this page causes the active list to be too large, it moves pages from the tail of the active list to the head of the inactive list. When the process exceeds its main memory allocation, the VM removes a page from the tail of the inactive list and evicts it to disk. This page is then inserted at the head of a third segment, the *evicted list*. When an address space's WSS exceeds its main memory allocation, the evicted list's histogram data allows the VM to project how large the allocation must be to capture the working set.

The active list is managed using a CLOCK algorithm. The inactive list is ordered by each page's time of removal from the active list. The relative sizes of these two lists is controlled by an adaptive mechanism described in Section 4.4. Like a traditional SEGQ, all inactive pages have their access permissions removed, forcing any reference to an inactive page to cause a minor page fault. When such a page fault occurs, the VM restores the page's permissions and promotes it into the active list, and then updates the address space's histogram. The insertion of a new page into the active list may force other pages out of the active list. The VM manages the evicted list similarly; the only difference is that a reference to an evicted page triggers disk activity.

4.2.1 Page Replacement Algorithm

The CRAMM VM places each `mem_info` structure into one of two lists: the *unused list* for the address spaces of files for which there are no open file descriptors, and the *normal list* for all other address spaces. When the VM must replace a page, it preferentially selects a `mem_info` from the unused list and then reclaims a page from the tail of that inactive list. If the unused list is empty, the VM selects a `mem_info` in a round robin manner from the normal list, and then selects a page from the tail of its inactive list.

As Section 5.2 will show, this eviction algorithm is less effective than the standard Linux VM replacement algorithm due to the round robin visiting of all processes. However, the CRAMM VM structure can support standard replacement policies and algorithms while also presenting the possibility of new policies that control per-address-space main memory allocation explicitly.

4.2.2 Available Memory

A garbage collector will periodically request that the CRAMM VM report the *available memory*—the total main memory space that could be allocated to the process. Specifically, the CRAMM VM reports the available memory (*available*) as the sum of the process's resident set size (*rss*), the free main memory (*free*), and the total number of pages found in the unused list (*unused*). There is also space reserved by the VM (*reserved*) to maintain a minimal pool of free pages that must be subtracted from this sum:

$$available = rss + free + unused - reserved$$

This value is useful to the collector because the CRAMM VM's per-address-space structure allows it to allocate this much space to a process without causing any page swapping. Standard VM systems that use global memory management (e.g., Linux) cannot identify the unused file space or preclude the possibility of page swapping as memory is re-allocated to a process.

4.3 Calculating Working Set Size

The CRAMM VM tracks the current working set size of each process. Recall that the WSS is *the smallest main memory allocation for which page faulting degrades process throughput by less than $t\%$.* If $t = 0$, space may be wasted by caching pages that receive very little use. When t is small but non-zero, the WSS may be substantially smaller than for $t = 0$, yet still yield only trivial page swapping. In our experiments, we chose $t = 5\%$.[2]

In order to calculate the WSS, the VM maintains an LRU reference histogram h [116, 133] for each process. For each reference to a page at position i of the process's page

[2] The threshold $t\%$ in our WSS definition indicates the overhead caused by page swapping when running an application. Usually a small but non-zero $t\%$ will give a mainingful WSS. We chose 5% because it is the level widely considered as acceptible in industry.

lists, the VM increments $h[i]$.[3] This histogram allows the VM to calculate the number of page faults that would occur for each possible memory allocation. The VM also monitors the mean cost of a major fault (*majfc*) and the time T that each process has spent on the CPU. To calculate the WSS, it scans the histogram backward to find the allocation at which the number of page faults is just below $(T \times t)/majfc$. The VM calculates the WSS of a GCed application at the end of each garbage collection, if it receives a notification from the garbage collector.

4.3.1 Page List Position

When a page fault occurs, the referenced page is found within the page lists using a hash map. In order to maintain the histograms, the CRAMM VM must determine the position of that page within the page lists. Because a linear traversal of the lists is inefficient, the VM attaches an AVL tree to each page list. Figure 4.1 shows this structure that the VM uses to calculate page list positions in logarithmic time. Specifically, every leaf node in the AVL tree points to a linked list of up to *k* pages, where *k* depends on the list into which the node points. Every non-leaf node is annotated with the total number of pages in its subtree; additionally, each non-leaf node is assigned a capacity that is the *k*-values of its children. The VM puts newly added pages into a buffer, and inserts this buffer into the AVL tree as a leaf node when that buffer points to *k* pages. Whenever a non-leaf node drops to half full, the VM merges its children and adjusts the tree shape accordingly.

When a page is referenced, the VM first searches linearly to find the page's position in the containing leaf node. It then walks up the AVL tree, summing the pages in leaf nodes that point to earlier portions of the page list. Thus, given that *k* is constant and small,

[3] Notice that we refer to the histogram as an *LRU reference histogram*, but that our page lists are not in true LRU order, and so the histogram is really a *SegQ reference histogram*. Also, note that only references to the inactive and evicted lists are applicable here, since references to active pages occur without kernel intervention.

determining a page's list position is performed in time proportional to the height of the AVL tree.

Because the CRAMM VM does not track references to pages in the active list, one leaf node contains pointers to all pages in the active list, and for this leaf node, $k = \infty$. For leaf nodes that point to inactive and evicted pages. We set k to 64—a value chosen to balance the work of linear search and tree traversal. The AVL trees have low space overhead. Suppose an application has N 4KB pages, and our AVL node structure is 24 bytes long. Here, the worst case space overhead (all nodes half full, so the total number of nodes is twice the number of leaf nodes) is:

$$\frac{((\frac{N}{64} \times 2 \times 2) \times 24)}{(N \times 2^{12})} < 0.037\%$$

On average, we observe that the active list contains a large portion (more than half) of the pages used by a process, and thus the observed space overhead is even lower.

4.3.2 LRU Histogram

Keeping one histogram entry for every page list position would incur a large space overhead. Instead, the CRAMM VM groups positions into *bins*. In our implementation, every bin corresponds to 64 pages (256 KB given the page size of 4 KB). This granularity is fine enough to provide a sufficiently accurate WSS measurement while reducing the space overhead substantially.

Furthermore, CRAMM dynamically allocates space for the histogram in chunks of 512 bytes. Given that a histogram entry is 8 bytes in size, one chunk corresponds to histogram entries for 16 MB of pages. Figure 4.2 shows the data structure for a histogram. We see that, when a process or file uses less than 64 pages (256 KB), it uses only bin_0, requiring no extra bins. This approach is designed to handle the frequent occurrence of small processes and files. Any process or file that requires more than 256 KB but less than 16MB memory uses the *level_1* histogram. Larger ones use the *level_2* histogram. The worst-case histogram

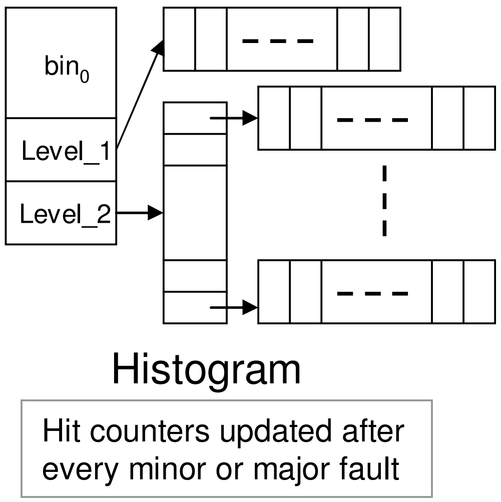


Figure 4.2. The structure of histogram

space overhead occurs when a process uses exactly 65 pages. Here, the histogram will need about 0.2% of the memory consumed by the process. In common cases, it is about 8 bytes per 64 pages, which is less than 0.004%.

4.3.3 Major Fault Cost

Calculating WSS requires tracking the mean cost of a major page fault. The CRAMM VM keeps a single, system-wide estimate of the mean fault cost, *majfc*. When the VM initiates a swap-in operation, it marks the page with a time-stamp. After the read completes, the VM calculates the time used to load the page. This new time is then used to update *majfc*.

4.4 Controlling Histogram Collection Overhead

Because the CRAMM VM updates a histogram entry at every reference to an inactive page, the size of the inactive list determines the overhead of histogram collection. If the inactive list is too large, then too much time will be spent handling minor page faults and updating histogram entries. If the inactive list is too small, then the histogram will provide too little information to calculate an accurate WSS. Thus, we want the inactive list to be as large as possible without inducing too much overhead. In our experiments, we attempt to control the overhead of updating histogram entries to be around 1%. This is small enough not to disturb the execution, but allows the CRAMM VM to collect enough information to calculate the application's WSS even in the face of dramatic phase changes.

The VM sets a target for *minor fault overhead*, expressed as a percentage increase in running time for processes, and dynamically adjusts the inactive list size according to this target. For each process, the VM tracks its CPU time T and a count of its minor page faults n. It also maintains a system-wide minor fault cost *minfc* using the same approach as with *majfc*. It uses these values to calculate the minor fault overhead as: $(n \times minfc)/T$. It performs this calculation periodically, after which it resets both T and n. Given a target of 1% and a constant threshold for deviation from that target of 0.5%, one of three cases may apply:

- If the overhead exceeds 1.5%, the VM decreases the inactive list size.
- If the overhead is less than 0.5%, it increases the inactive list size.
- If there are no minor faults during this period, and if the inactive list is not full, then it moves pages from the active to the inactive list (*refilling* the inactive list).

This simple adaptive mechanism, set to a 1% overhead target and a 0.5% deviation threshold, successfully keeps the overhead low while yielding sufficient histogram information for WSS calculations.

4.4.1 Inactive List Size Adjustment

CRAMM assigns each process a *target inactive size*, initially 0. When CRAMM adjusts the inactive list size, it is really setting this target size. Assume that a process has P_A pages in the active list and P_I in the inactive list. Depending on the overhead's relationship to its threshold, the new target will be:

- Increase: $P_I + max(min(P_A, P_I)/32, 8)$
- Decrease: $P_I - max(min(P_A, P_I)/8, 8)$
- Refill: $P_I + max(min(min(P_A, P_I)/16, 256), 8)$

By choosing the smaller of P_A and P_I in these equations, we make the adjustments small if either list is small, thus not changing the target too drastically. These formulas also ensure that at least some constant change is applied to the target, ensuring a change that will have some effect. We also put an upper bound on the refilling adjustment to prevent flushing too many pages into the inactive list at a time. Finally, we decrease the target inactive list size more aggressively than we increase it because low overhead is a more critical and sensitive goal than accurate histogram information. We also refill more aggressively than we increase because zero minor faults is a strong indication of an inadequate inactive list size.

Whenever a page is added to the active list, the VM checks the current inactive list size. If it is less than its target, then the VM moves several pages from the active list to the inactive list (8 pages in our implementation). When an adjustment triggers refilling, the VM immediately forces pages into the inactive list to match its new target.

4.4.2 Adaptivity Triggers

In the CRAMM VM, there are two events that can trigger an inactive list size adjustment. The first, *adjust_interval*, is based on running time, and the second, *adjust_count*, is based on the number of minor faults.

For every new process, its *adjust_interval* is initialized to a default value ($\frac{1}{16}sec$). Whenever a process is scheduled, if its running time since the last adjustment exceeds its *adjust_interval* value, then the VM adjusts the inactive list size.

The *adjust_count* variable is initialized to be $(adjust\_interval \times 2\%)/minfc$. If a process suffers this number of minor faults before *adjust_interval* CPU time has passed, then its overhead is well beyond the acceptable level. At each minor fault, the VM checks whether the number of minor faults since the last adjustment exceeds *adjust_count*. If so, it forces an adjustment.

CHAPTER 5

CRAMM: EXPERIMENTAL EVALUATION

We implemented the CRAMM VM system in the Linux kernel and the CRAMM heap sizing model in the Jikes RVM research Java virtual machine [6], and applied our heap size manager to five different garbage collectors. We now evaluate our CRAMM VM implementation and heap size manager. We first compare the performance of the CRAMM VM with the original Linux VM. We then add the heap size manager to several collectors in Jikes RVM, and evaluate their performance under both static and dynamic real memory allocations. We also compare them with the JRockit [19] and HotSpot [69] JVMs under similar conditions. Finally, we run two concurrent instances of our adaptive collectors under memory pressure to see how they interact with each other. These results demonstrate CRAMM's effectiveness in maintaining high performance in the face of changes in application behavior and system load.

5.1 Methodology Overview

We performed all measurements on a 1.70GHz Pentium 4 Linux machine with 512MB of RAM and 512MB of local swap space. The processor has 12KB I and 8KB D L1 caches and a 256KB unified L2 cache. We installed both the "stock" Linux kernel (version 2.4.20) and our CRAMM kernel. We ran each of our experiments six times in single-user mode, and always report the mean of the last five runs. In order to simulate memory pressure, we used a background process to pin a certain volume of pages in memory using `mlock`.

5.1.1 Application Platform

We used Jikes RVM v2.4.1 [6] built for Linux x86 as our Java platform. We optimized the system images to the highest optimization level to avoid run-time compilation of those components. Jikes RVM uses an *adaptive* compilation system, which invokes optimization based on time-driven sampling. This makes executions non-deterministic. In order to get comparable deterministic executions, we took compilation logs from 7 runs of each benchmark using the adaptive system, and directed the system to compile methods according to the log from the run with the best performance. This is called the *replay* system. It is deterministic and highly similar to typical adaptive system runs.

5.1.2 Garbage Collectors

We evaluate five collectors from the MMTk memory management toolkit [23] in Jikes RVM: MS (mark-sweep), GenMS (generational mark-sweep), CopyMS (copying mark-sweep), SS (semi-space), and GenCopy (generational copying). All of these collectors have a separate non-copying region for large objects (2KB or more), collected with the Treadmill algorithm [10]. They also use separate non-copying regions for meta-data and immortal objects. We now describe the *other* regions each collector uses for ordinary small objects. MS is non-generational with a single MS region. GenMS is generational with a copying nursery and MS mature space. CopyMS has two regions, both collected at every GC. New objects go into a copy region, while copy survivors go into an MS region. SS is non-generational with a single copying region. GenCopy is generational with copying nursery and mature space. Both generational collectors (GenMS and GenCopy) use Appel-style nursery sizing [8] (starts large and shrinks as mature space grows).

5.1.3 Benchmarks

For evaluating JVM performance, we use a wide ranges of benchmarks. We ran all benchmarks from the SPECjvm98 suite (standard and widely used), plus those benchmarks from the DaCapo suite [24, 25] (an emerging standard for JVM GC evaluation) that run under Jikes RVM, plus `ipsixql` (a publicly available XML database program) and `pseudojbb` (a variant of the standard, often-used SPECjbb server benchmark with a fixed workload (140,000 transactions) instead of fixed time limit). For evaluating general VM performance, we used the standard SPEC2000 suite.

Most results are similar, so to make our presentation clear we present results from some representative collectors and benchmarks that cover all interesting cases. For collectors, we chose SS, MS, and GenMS to cover copying, non-copying, and generational variants. For benchmarks, we chose `javac`, `jack`, `pseudojbb`, `ipsixql`, `jython`, and `pmd` to cover all the benchmark suites we used.

5.2 CRAMM VM Performance

For the CRAMM VM to be practical, its baseline performance (i.e., while collecting useful histogram/working set size information) must be competitive with a standard VM when physical RAM is plentiful.

We first compare the performance of the CRAMM VM to that of the stock Linux kernel across our entire SPEC2000 benchmark suite.[1] In order to get stable results, we use the input that makes each benchmark run longer than 60 seconds. Table 5.1 presents the results of these experiments, with integer benchmarks (INT) on the left and floating point benchmarks (FP) on the right. For each program we give the percentage increase in execution time under CRAMM versus the stock Linux kernel (I), the percentage of time spent handling minor faults (M), the total execution time (T), and the number of minor faults trapped by CRAMM kernel. While the inactive list size adjustment mechanism effectively keeps the cost of collecting histogram data in the desired range (e.g., 0.59% for SPEC2Kint and 1.02% for SPEC2Kfp), the slowdown is generally about 1.0–3.0%. We believe this overhead is caused by CRAMM polluting the cache when handling minor faults as it processes

[1] We could not compile and run some SPEC2000 Fortran programs, so we omit these FP benchmarks.

INT	I(%)	M(%)	T(s)	faults	FP	I(%)	M(%)	T(s)	faults
gzip	2.17	0.66	93.11	101260	wupwise	1.96	0.88	260.25	420273
vpr	0.95	0.23	228.06	99447	swim	0.58	0.72	522.13	627641
gcc	1.57	0.50	78.51	52566	applu	3.08	2.78	501.80	2608207
mcf	1.34	0.74	367.12	331030	mgrid	2.01	1.69	417.58	1369456
crafty	0.36	0.34	227.02	131825	mesa	0.98	1.01	449.24	815891
parser	0.50	0.45	574.89	377250	art01	0.78	0.43	493.50	425467
perlbmk	0.57	0.54	74.23	43631	equake	0.89	0.75	218.42	297999
gap	1.33	1.22	183.15	398268	ammp	1.06	0.97	782.01	1317055
vortex	2.12	0.55	111.42	61453	apsi	2.20	0.73	961.49	1136198
bzip2	0.90	0.66	130.69	131630	twolf	-0.62	0.20	808.00	311724
mean	1.18	0.59			mean	1.29	1.02		

Table 5.1. CRAMM VM Performance (SPEC2000 INT & FP)

Benchmark	GenCopy		SemiSpace		MarkSweep		GenMS		CopyMS	
	I(%)	M(%)	I(%)	M(%)	I(%)	M(%)	I(%)	M(%)	I(%)	M(%)
compress	1.70	0.82	0.97	0.89	1.42	0.88	0.29	0.91	2.91	0.91
jess	1.48	0.38	2.98	1.28	2.58	1.37	3.94	1.49	3.13	1.48
raytrace	1.43	0.49	2.29	1.24	2.48	1.37	3.05	1.42	2.47	1.34
db	1.35	0.83	1.46	1.01	1.49	1.00	1.49	0.86	1.50	0.97
javac	2.79	1.01	2.30	1.39	2.91	1.50	2.00	1.20	2.13	0.99
mpegaudio	0.56	0.33	0.63	0.37	1.57	0.94	1.66	0.94	1.55	0.81
mtrt	0.84	0.43	2.53	1.14	2.67	1.27	2.82	1.27	2.54	1.32
jack	1.90	0.61	2.85	1.21	1.98	1.25	3.15	1.41	2.30	1.23
pseudojbb	0.42	1.08	1.20	1.32	1.16	1.08	1.81	1.06	1.67	1.05
ipsixql	2.28	0.89	2.70	1.37	4.39	1.14	5.31	1.18	4.89	1.16
mean	**1.48**	**0.69**	**1.99**	**1.12**	**2.27**	**1.18**	**2.55**	**1.17**	**2.51**	**1.13**

Table 5.2. CRAMM VM Performance (SPECjvm98, `pseudojbb`, and `ipsixql`)

page lists and AVL trees. This, in turn, leads to extra cache misses for the application. We verified that at the target minor fault overhead, CRAMM incurs enough minor faults to calculate the working set size accurately with respect to our 5% page fault threshold.

We further evaluate the performance of the CRAMM VM using Java benchmarks under five garbage collectors. Table 5.2 presents the results, showing only the percentage increase in execution time (I) and percentage of time spent in handling minor faults (M). These results are strongly similar to those of SPEC2000 benchmarks. We notice that CRAMM's performance is generally somewhat poorer on the Java benchmarks, where it must spend more time handling minor faults caused by the dramatic working set changes between the

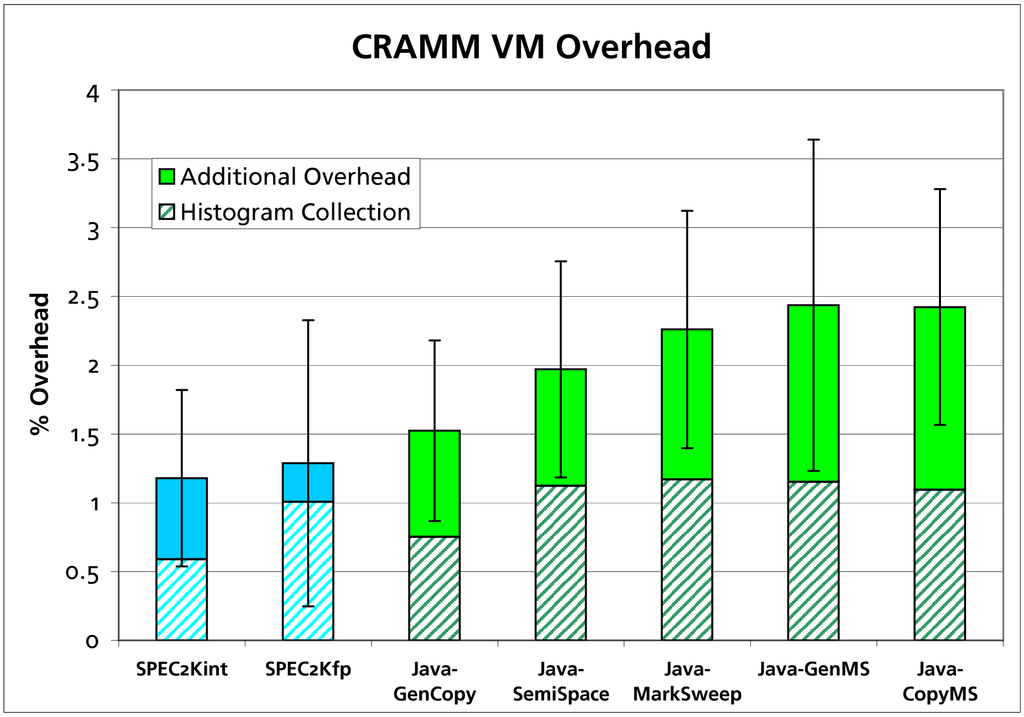


Figure 5.1. Virtual memory overhead (% increase in execution time) without paging, across all benchmark suites and garbage collectors.

mutator and collector phases of GCed applications. However, the fault handling overhead remains in our target range.

Figure 5.1 summarizes all the experimental results, which are geometric means across all benchmarks: SPEC2000int, SPEC2000fp, and Java benchmarks (SPECjvm98, DaCapo, pseudojbb, and ipsixql) with five different garbage collectors. Overall, CRAMM collects the necessary information with low overhead in most cases (about 1–2.5%), and its performance is competitive to that of the stock kernel. In exchange for this small percentage overhead, the CRAMM VM enables our adaptive garbage collectors to select heap sizes predictively on the fly, reducing total execution time dramatically in the face of memory pressure.

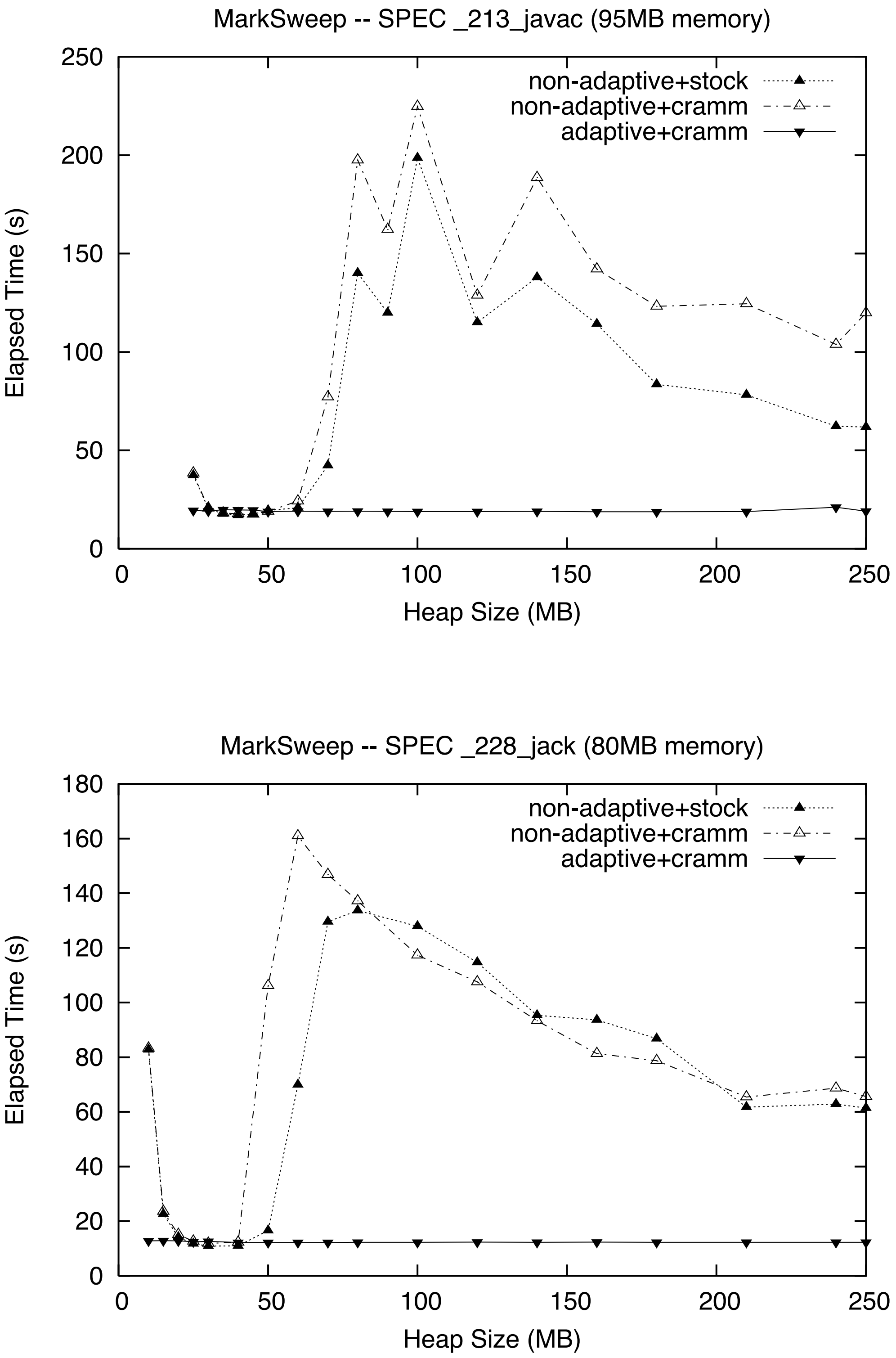
5.3 Static Memory Allocation

To test our adaptive mechanism, we run the benchmarks over a range of requested heap sizes with a fixed memory allocation. We select memory allocations that reveal the effects of large heaps in small allocations and small heaps in large allocations. In particular, we try to evaluate the ability of our mechanism to grow and shrink the heap. We run the non-adaptive collectors (which simply use the requested heap size) on both the stock and CRAMM kernels, and the adaptive collectors on the CRAMM kernel, and compare performance.

5.3.1 MarkSweep Collector

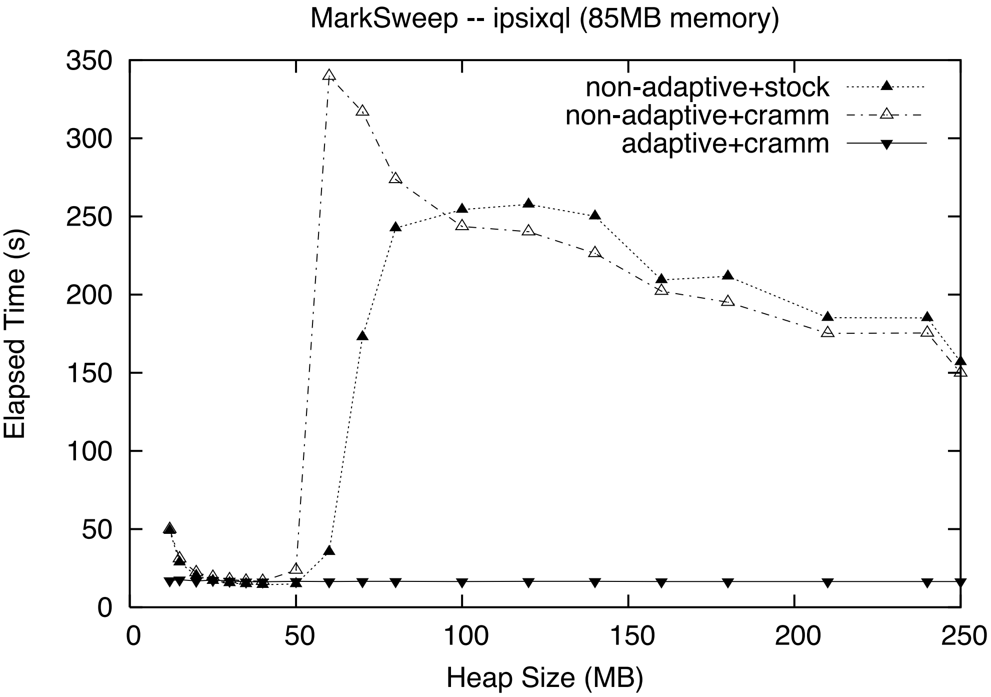
Figures 5.2, 5.3, and 5.4 present the execution time for benchmarks using the MS collector with a static memory allocation. For almost every combination of benchmark and requested heap size, our adaptive collector chooses a heap size that is nearly optimal. It reduces total execution time dramatically, or performs at least as well as the non-adaptive collector. At the leftmost side of each curve, the non-adaptive collector runs at a heap size that does not consume the entire allocation, thus under-utilizing available memory, collecting too frequently and inducing high GC overhead. The adaptive collector grows the heap size to reduce the number of collections without incurring paging. At the smallest requested heap sizes, this adjustment reduces execution time by as much as 85%.

At slightly larger requested heap sizes, the non-adaptive collector performs fewer collections, better utilizing available memory. One can see that there is an ideal heap size for the given benchmark and allocation. At that heap size, the non-adaptive collector performs well—but the adaptive collector often matches it, and is never very much worse. The maximum slowdown we observed is 11% across all the benchmarks. (Our working set size calculation uses a page fault threshold of $t = 5\%$, so we are allowing a trivial amount of paging—while reducing the working set size substantially.)


MarkSweep -- SPEC _213_javac (95MB memory)

MarkSweep -- SPEC _228_jack (80MB memory)

Figure 5.2. Static Memory Allocation: MarkSweep (`javac` and `jack`)



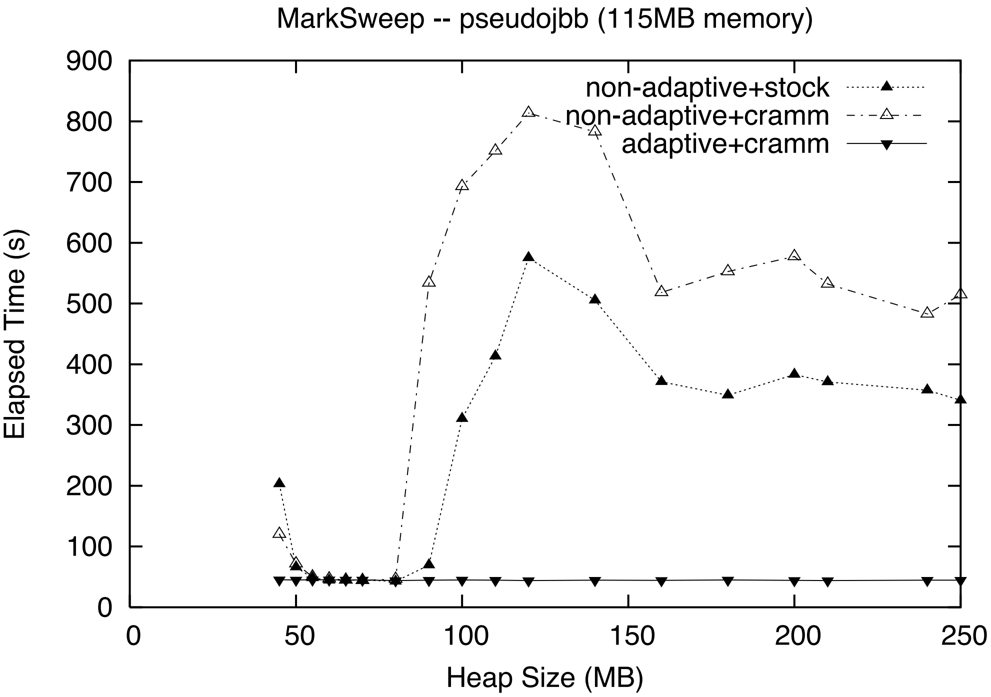
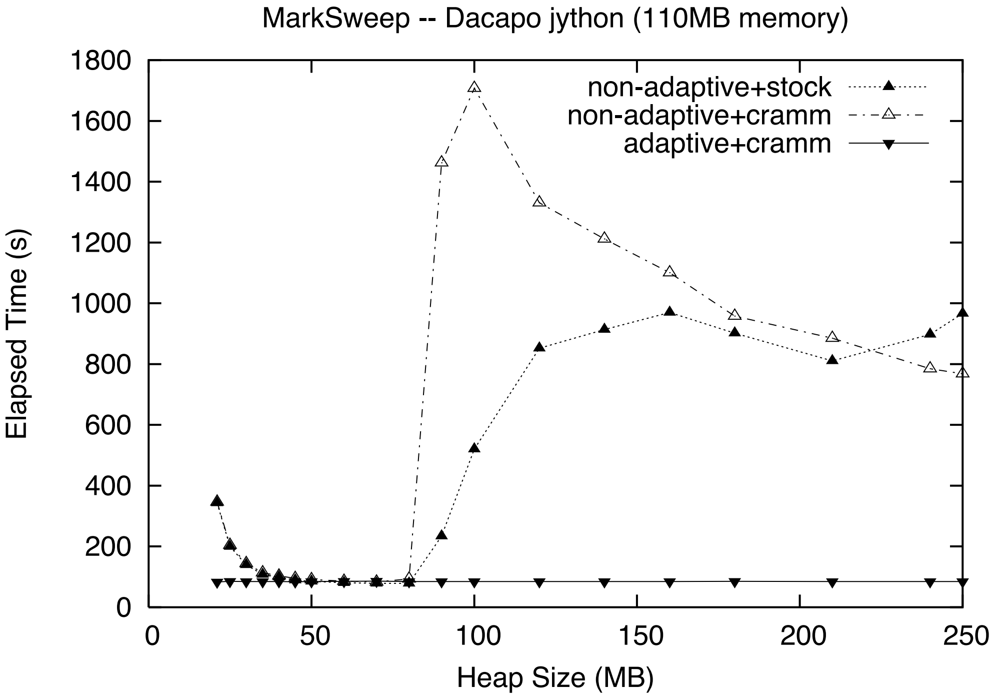


Figure 5.3. Static Memory Allocation: MarkSweep (`ipsixql` and `pseduojbb`)



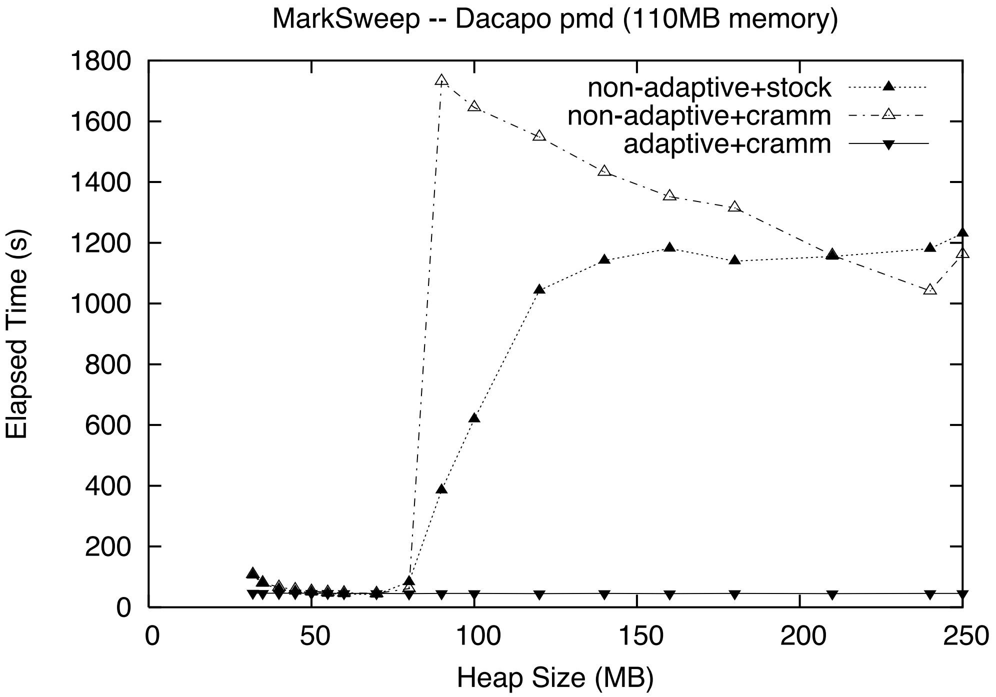
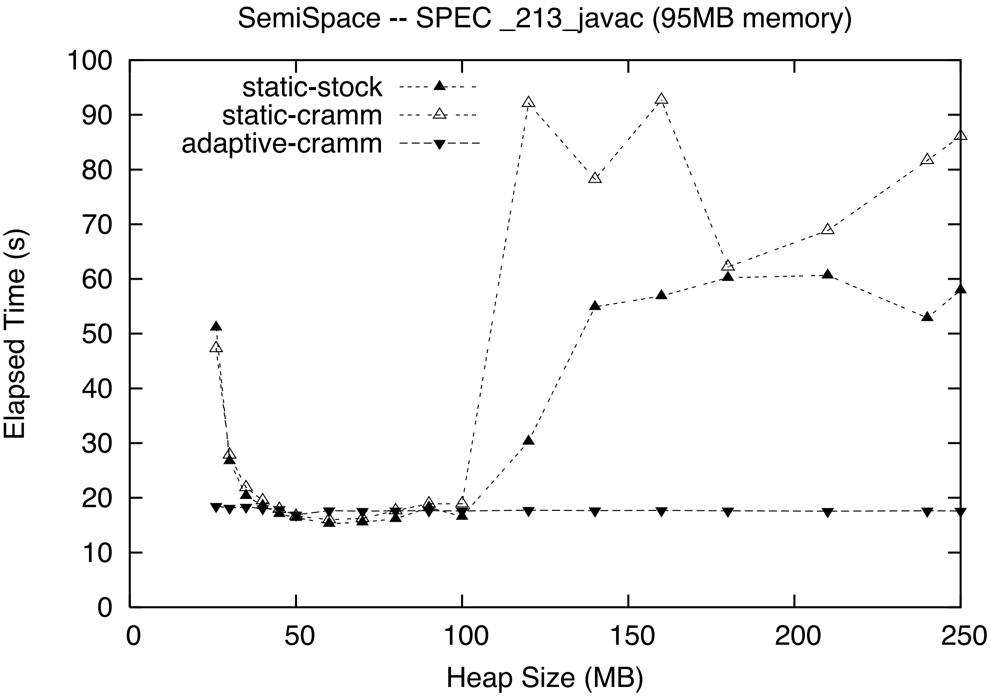


Figure 5.4. Static Memory Allocation: MarkSweep (`jython` and `pmd`)



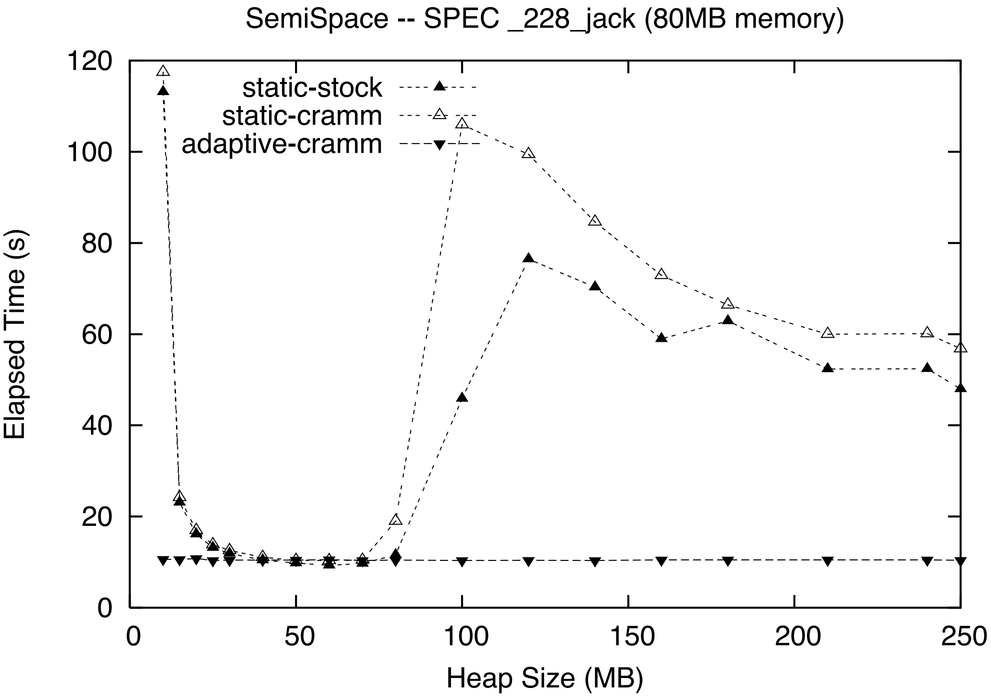


Figure 5.5. Static Memory Allocation: SemiSpace (`javac` and `jack`)

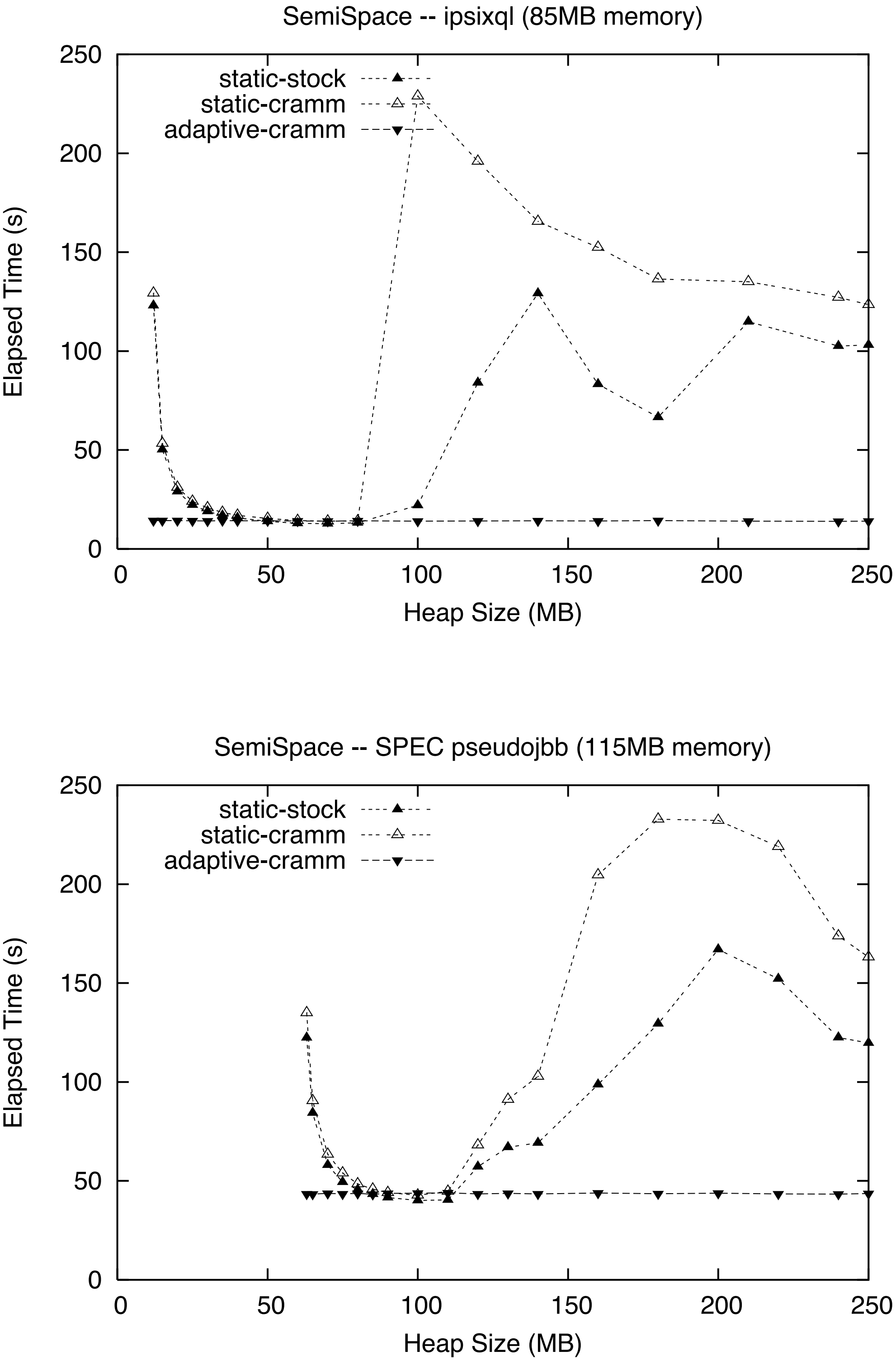


Figure 5.6. Static Memory Allocation: SemiSpace (`ipsixql` and `psuedojbb`)

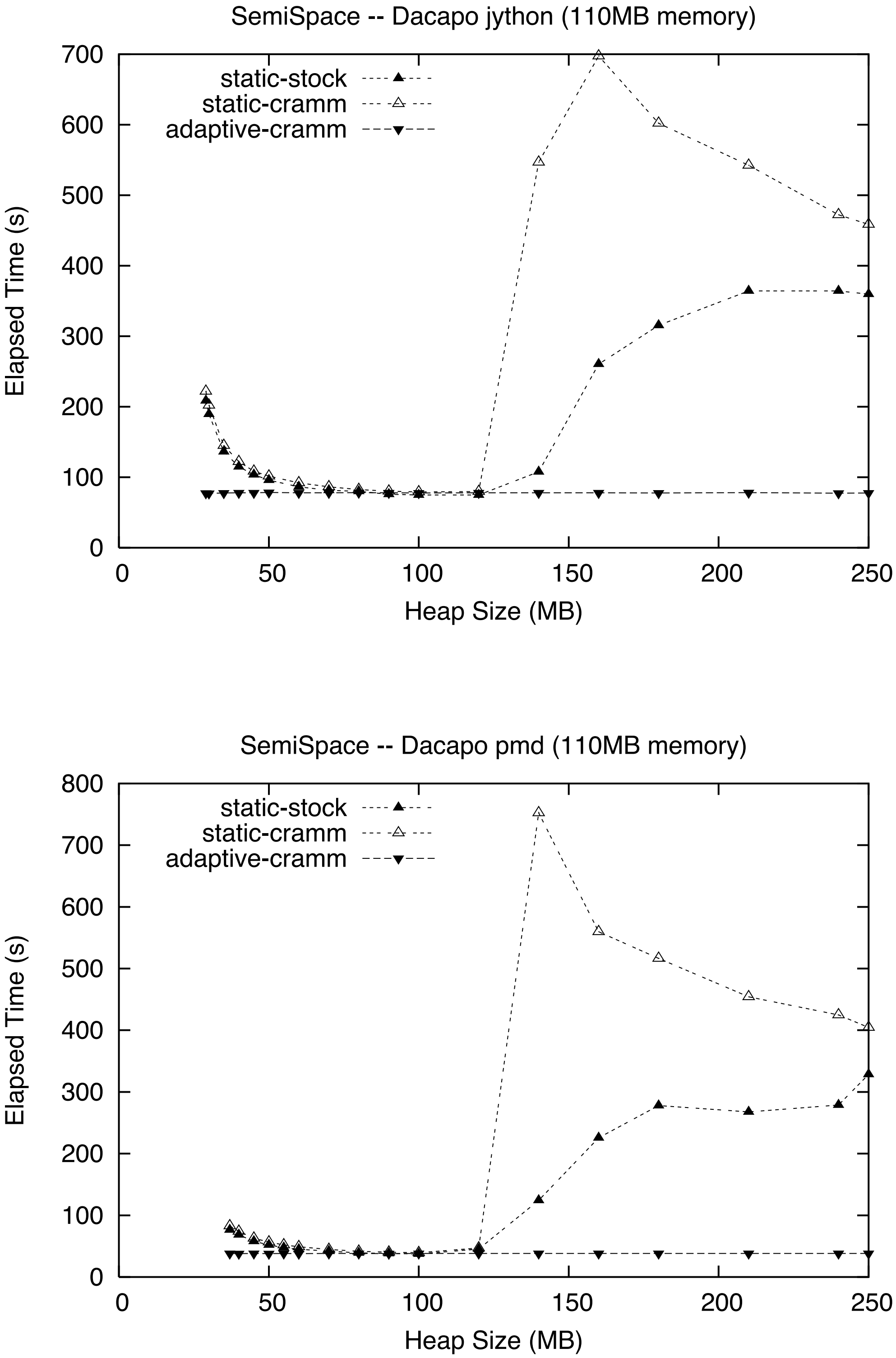
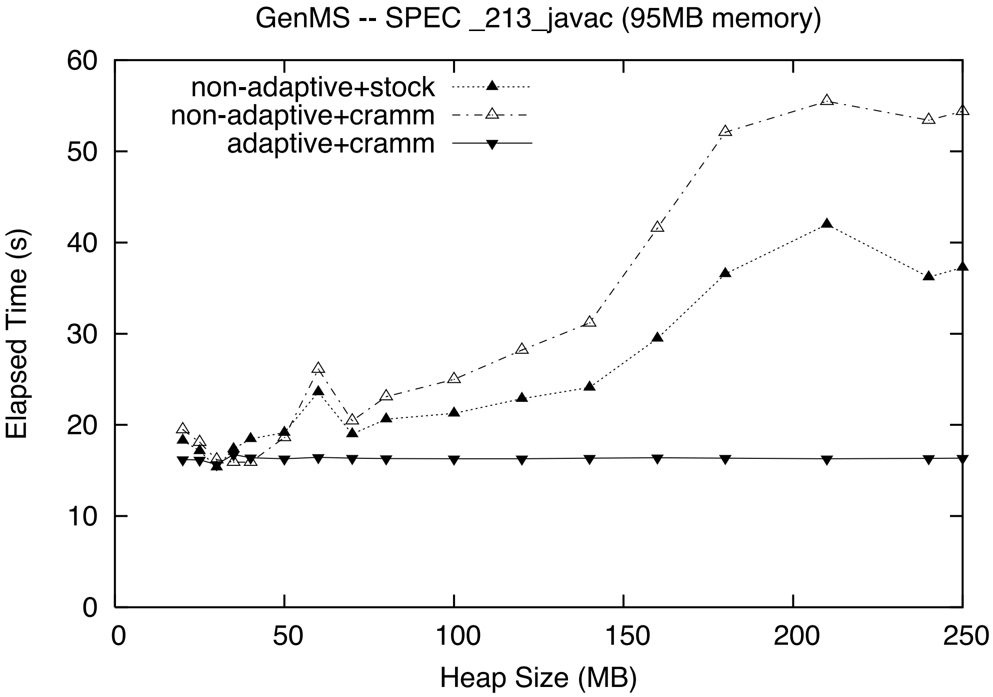


Figure 5.7. Static Memory Allocation: SemiSpace (`jython` and `pmd`)



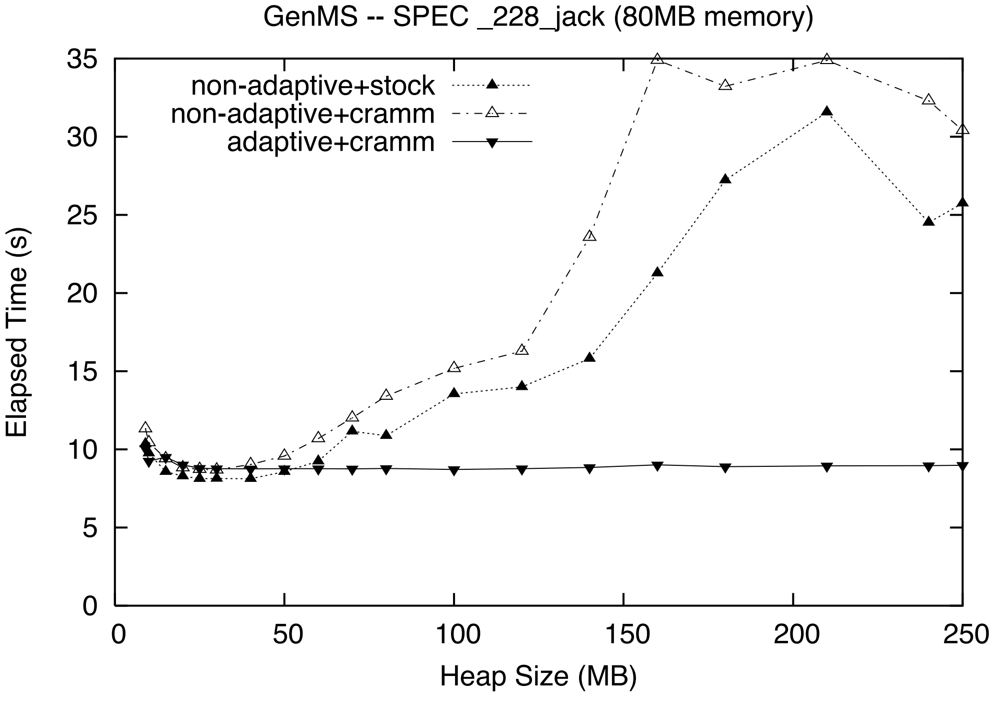
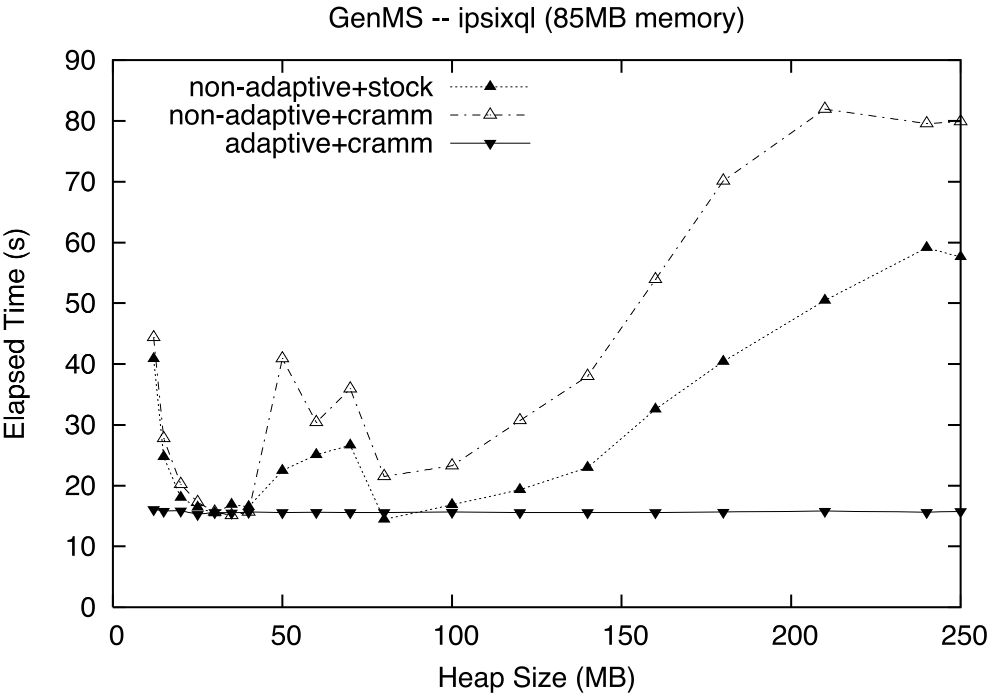


Figure 5.8. Static Memory Allocation: GenMS (`javac` and `jack`)



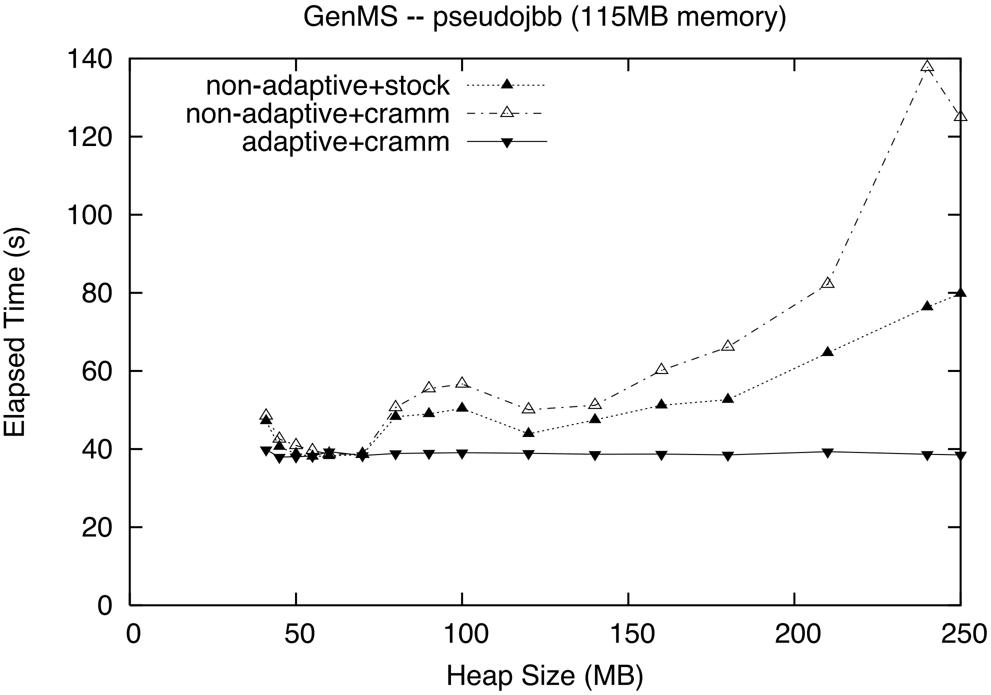
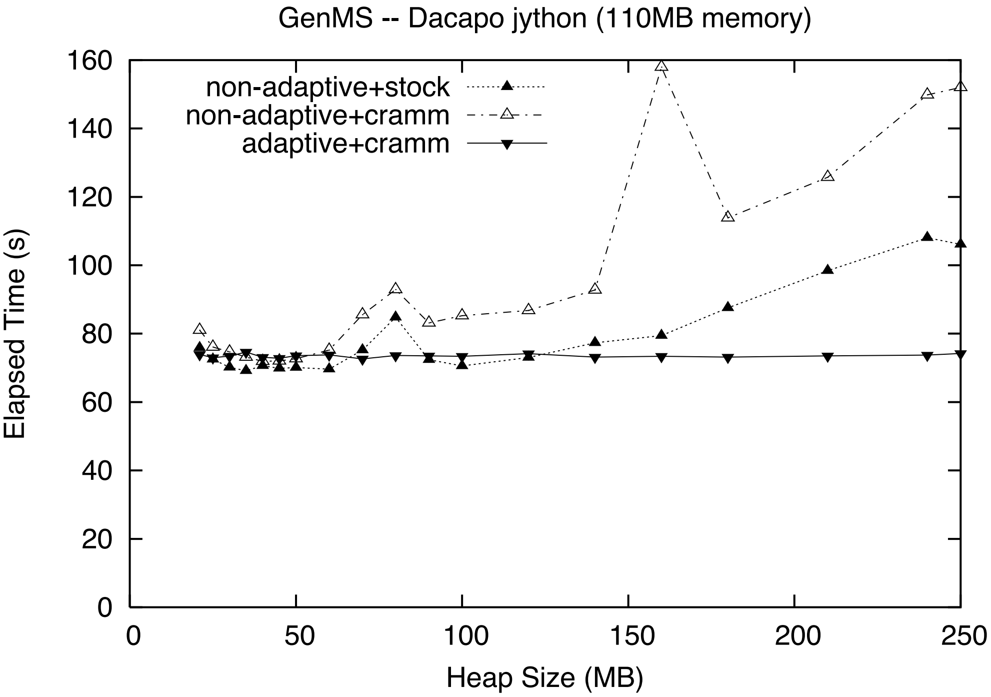


Figure 5.9. Static Memory Allocation: GenMS (`ipsixql` and `pseudojbb`)



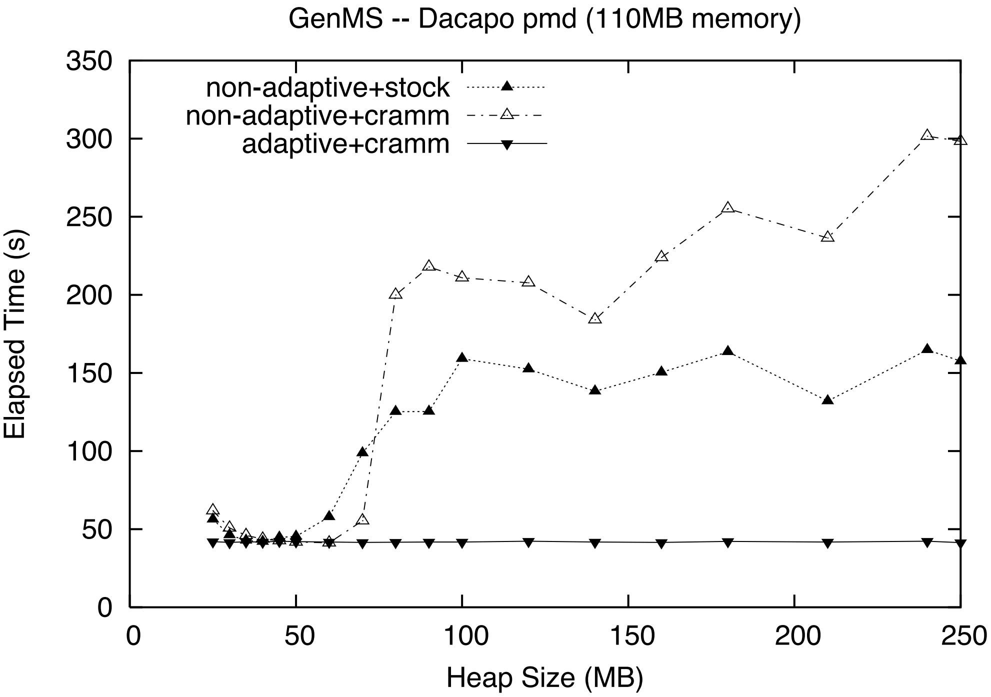


Figure 5.10. Static Memory Allocation: GenMS (`jython` and `pmd`)

Once the requested heap size goes slightly beyond the ideal, the non-adaptive collector's performance drops dramatically. The working set size is just slightly too large for the allocation, which induces enough paging to slow execution by as much as a factor of 5 to 10. In contrast, our adaptive collector shrinks the heap so that the allocation completely captures the working set size. By performing slightly more frequent collections, the adaptive collector consumes a modest amount of CPU time to avoid a lot of paging, thus reducing elapsed time by as much as 90%. When the requested heap size becomes even larger, the performance of our adaptive collector remains the same. However, the execution time of the non-adaptive collector decreases gradually. This is because it does fewer collections, and it is collections that cause most of the paging.

Interestingly, when we disable adaptivity, the CRAMM VM exhibits worse paging performance than the stock Linux VM. An LRU-based eviction algorithm turns out to be a poor fit for garbage collection's memory reference behavior. Collectors typically exhibit loop-like behavior when tracing live objects, and LRU is notoriously bad in handling large loops. The Linux VM instead uses an eviction algorithm based on a combination of CLOCK and a linear scan over the program's address space, which happens to work better in this case.

5.3.2 SemiSpace and GenMS Collector

Figures 5.5, 5.6, and 5.7 show the results of the same experiments for the SemiSpace collector (a copying collector). Figures 5.8, 5.9, and 5.10 are for the GenMS collector (a generational collector with a marksweep mature space). As we can see, these graphs are qualitatively highly similar to those for the MarkSweep collector. For almost every combination of benchmark and its corresponding given memory allocation, our adaptive collectors choose a heap size that is large enough to make full use of available memory, but small enough to avoid substantial paging, indicating that CRAMM's analytical heap sizing model indeed works very well with various garbage collectors.

Benchmark	Collector		Enough Mem		Adaptive			Non-Adaptive			Adaptive
(Memory)	(Heap Size)		T(sec)	MF	T(sec)	cpu	MF	T(sec)	cpu	MF	Yes/No
pseudojbb	SS	(160M)	297.35	1136	339.91	99%	1451	501.62	65%	24382	0.678
(160M)	MS	(120M)	336.17	1136	386.88	98%	1179	928.49	36%	47941	0.417
	GenMS	(120M)	296.67	1136	302.53	98%	1613	720.11	48%	39944	0.420
javac	SS	(150M)	237.51	1129	259.35	94%	1596	455.38	68%	24047	0.569
(140M)	MS	(90M)	261.63	1129	288.09	95%	1789	555.92	47%	25954	0.518
	GenMS	(90M)	249.02	1129	263.69	95%	2073	541.87	50%	33712	0.487

Table 5.3. Dynamic Memory Allocation: Performance of Adaptive vs. Non-Adaptive Collectors

5.4 Dynamic Memory Allocation

The results given so far show that our adaptive mechanism selects a good heap size when presented with an unchanging memory allocation. We now examine how CRAMM performs when the memory allocation changes dynamically. To simulate dynamic memory pressure, we use a background process that repeatedly consumes and releases memory. Specifically, it consists of an infinite loop, in which it sleeps for 25 seconds, `mmap`'s 50MB memory, `mlock`'s it for 50 seconds, and then unlocks and unmaps the memory. We also modify how we invoke benchmarks so that they run long enough (we give `pseudojbb` a large transaction number, and iterate `javac` 20 times).

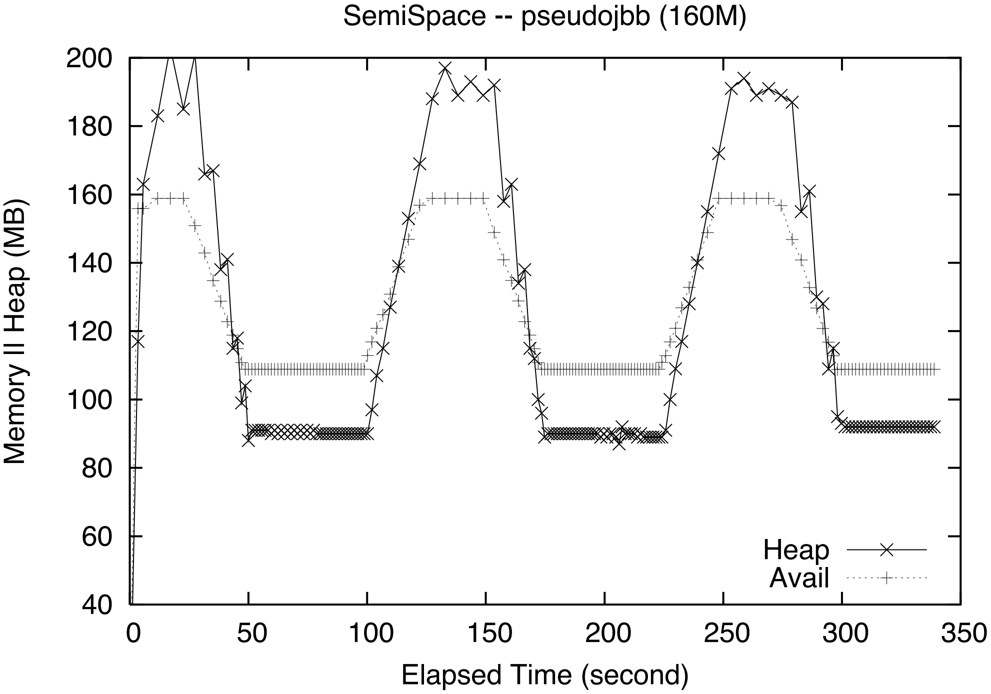
5.4.1 Adapting to Dynamic Memory Pressure

Table 5.3 summarizes the performance of both non-adaptive and adaptive collectors under this dynamic memory pressure. The first column gives the benchmarks and their initial memory allocation. The second column gives the collectors and their requested heap sizes respectively. We set the requested heap size so that the benchmark will run gracefully in the initial memory allocation. We present the total elapsed time (T), CPU utilization (cpu), and number of major faults (MF) for each collector. We compare them against the base case, i.e., running the benchmark at the requested heap size with sufficient memory. The last column shows adaptive execution time relative to non-adaptive. We see that for each collector the adaptive mechanism adjusts the heap size in response to memory pressure, nearly eliminating paging. The adaptive collectors show very high CPU utilization and dramatically reduced execution time.

Figures 5.11, 5.12, and 5.13 illustrate how our adaptive collectors (SemiSpace, MarkSweep and GenMS respectively) change the heap size while running `pseudojbb` under dynamic memory pressure. The upper graphs on each page demonstrate how available memory changes over time, and the corresponding heap size chosen by each adaptive collector. We see that as available memory drops, the adaptive collectors quickly shrink the heap to avoid paging. Likewise, they grow the heap responsively when there is more available memory. One can also see that the difference between the maximum and minimum heap size is approximately the amount of memory change divided by heap utilization u, which is consistent with our working set size model presented in Section 3.3.

We also compare the throughput of the adaptive and non-adaptive collectors (the lower graphs in Figures 5.11, 5.12, and 5.13), by printing out the number of transactions finished as time elapses for `pseudojbb`. These curves show that memory pressure has much less impact on throughput when running under our adaptive collectors. It causes only a small disturbance and only for a short period of time (i.e., during the period of dynamic memory pressure). The total execution time of our adaptive collectors is a little longer than that of the base case, simply because they run at a much smaller heap size (and thus collect more often) when there is less memory. The non-adaptive collectors experience significant paging slowdown when under memory pressure.

Figures 5.14, 5.15, and 5.16 demonstrate the heap adjustment performed by our adaptive collectors while running `javac` under the same dynmic memory pressure. They are qualitatively similar to those for `pseudojbb`, with one special case, where the SemiSpace collector chooses an unusually small heap size. Our analysis of the execution trace shows that this is a pretty interesting case, caused by our iterative execution of `javac`. This is because that garbage collection happened to be performed during the initialization stage of one iteration, resulting in an exceptionally high survival rate. Our adaptive collector



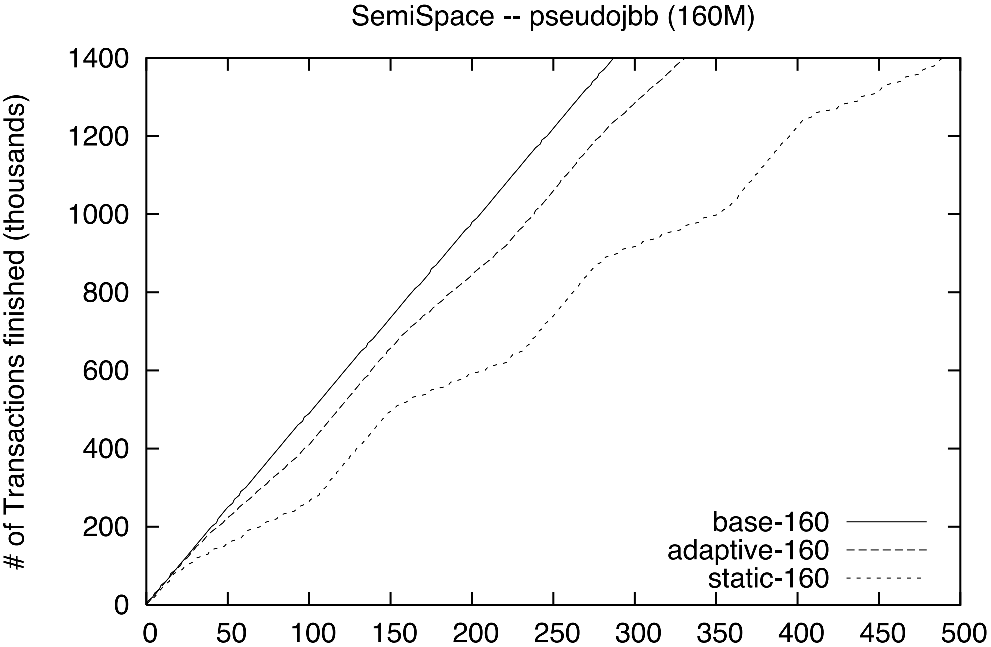


Figure 5.11. Dynamic Memory Allocation (SemiSpace, `pseudojbb`): Heap Adjustment and Throughput

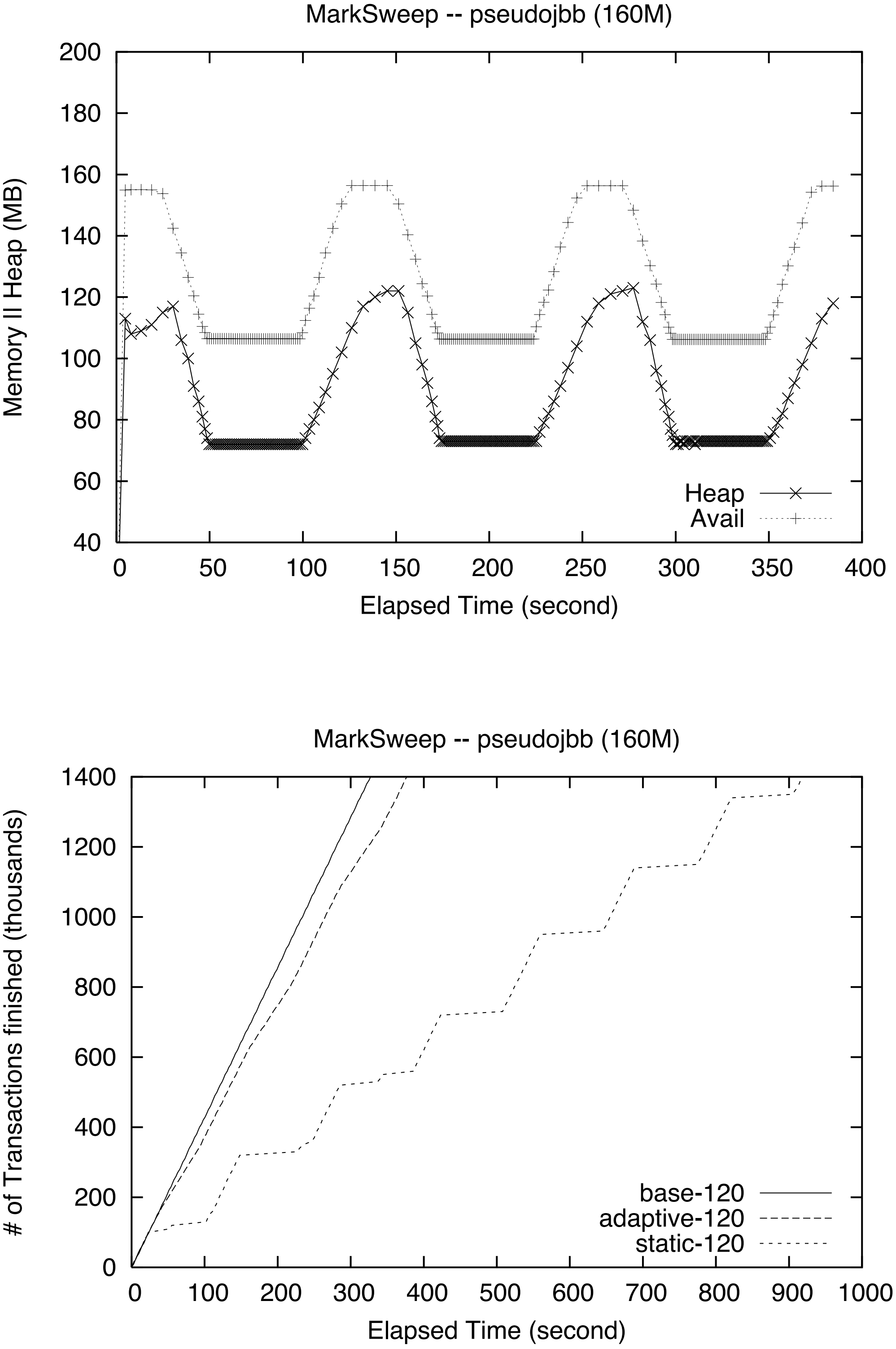
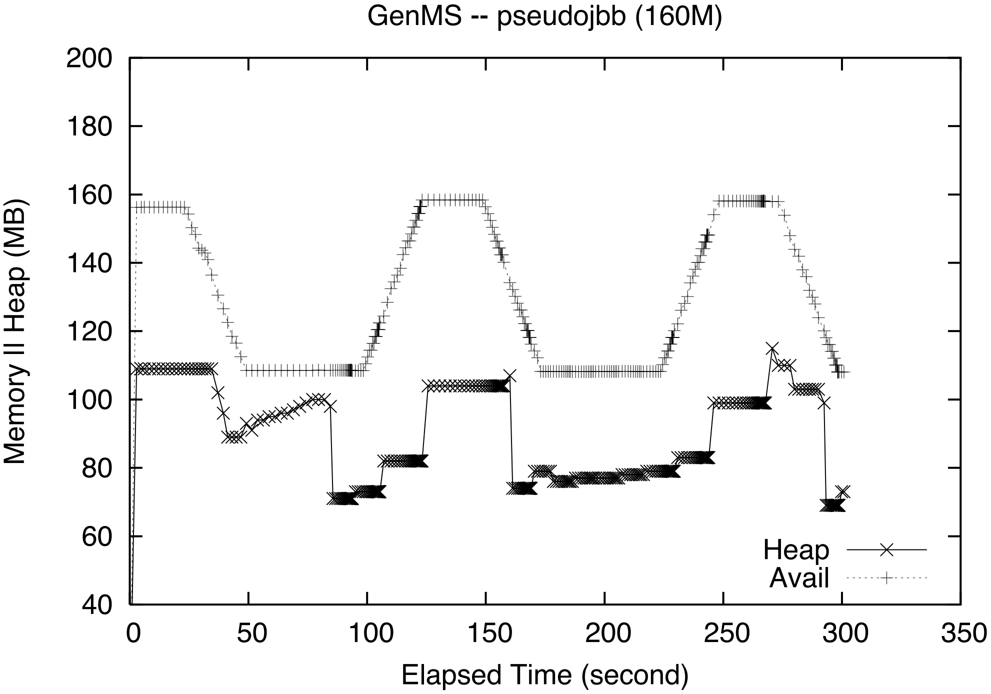


Figure 5.12. Dynamic Memory Allocation (MarkSweep, `pseudojbb`): Heap Adjustment and Throughput



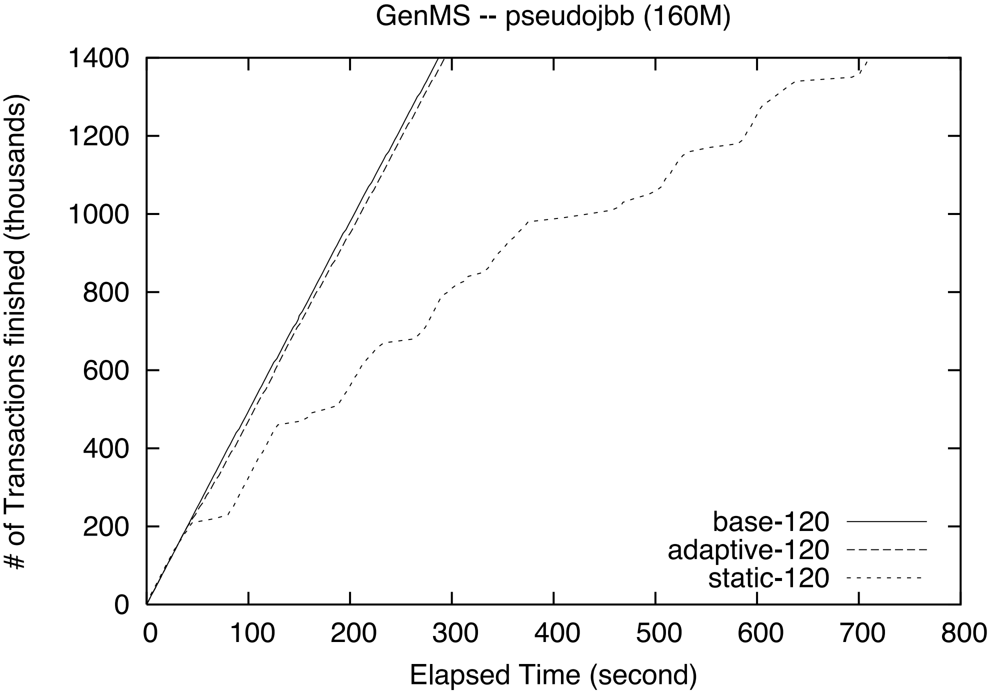


Figure 5.13. Dynamic Memory Allocation (GenMS, `pseudojbb`): Heap Adjustment and Throughput

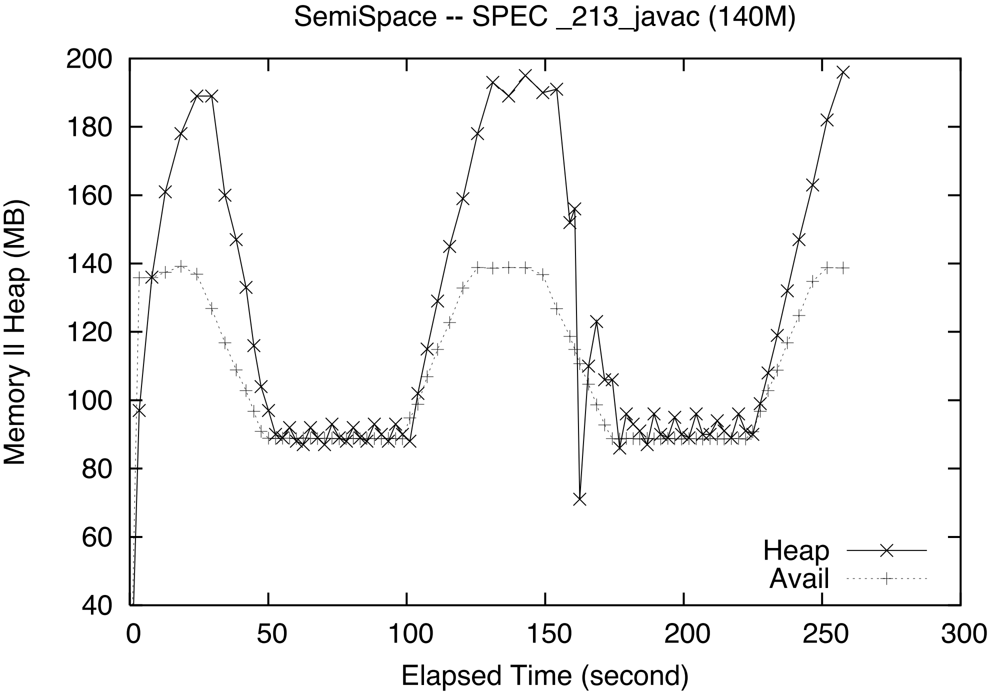


Figure 5.14. Dynamic Memory Allocation (SemiSpace, `javac`): Heap Adjustment

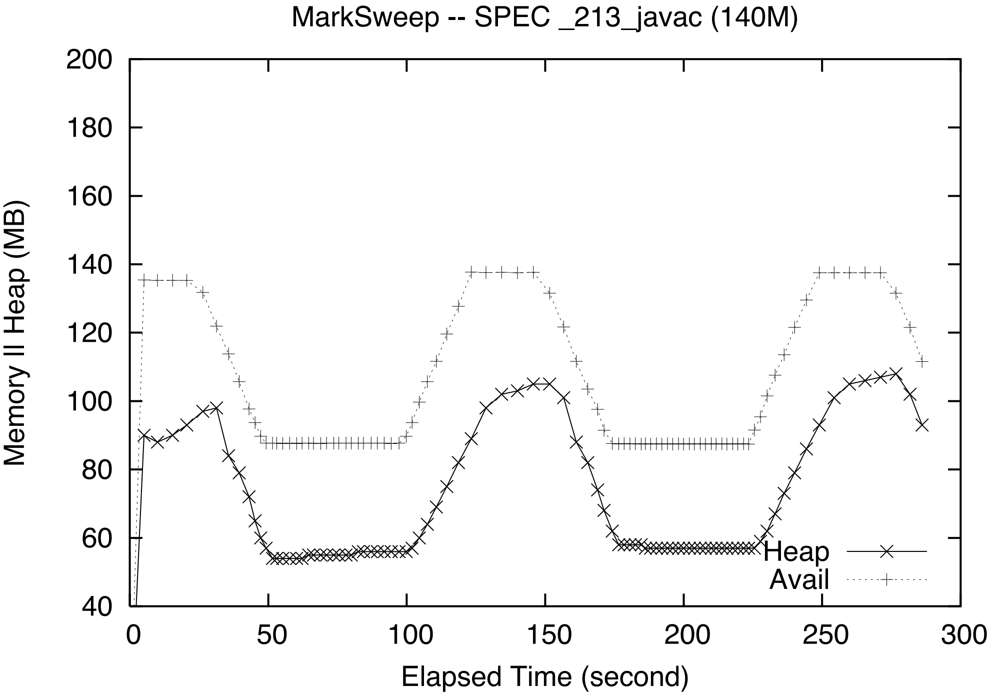


Figure 5.15. Dynamic Memory Allocation (MarkSweep, `javac`): Heap Adjustment

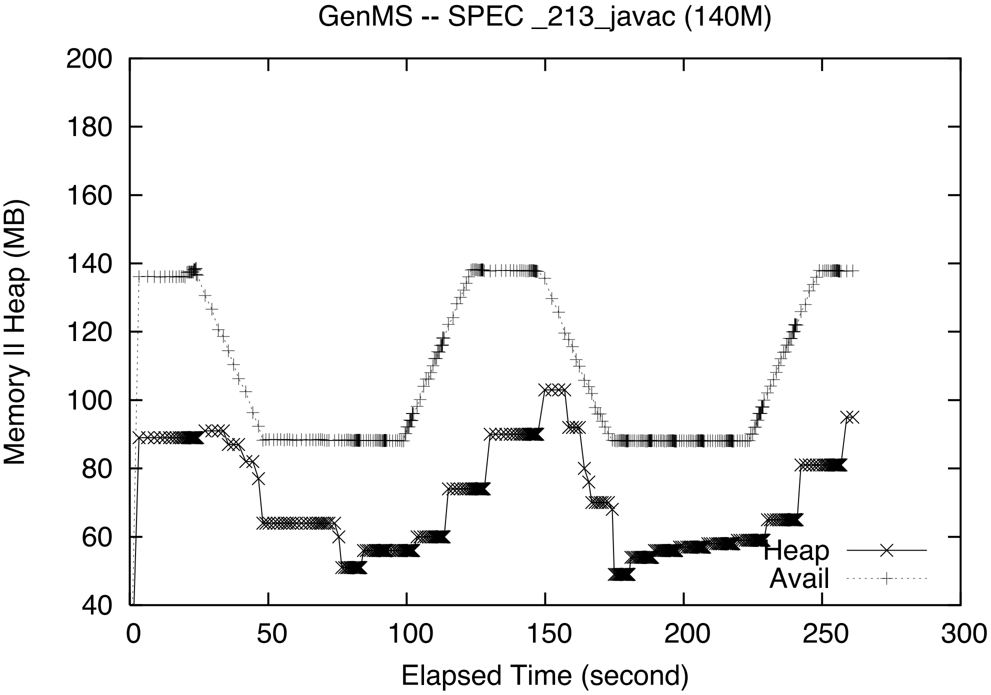


Figure 5.16. Dynamic Memory Allocation (GenMS, `javac`): Heap Adjustment

thus chose a smaller heap size to avoid paging, assuming that the amount of copy survived objects would continue to increase during next collection. However, it decayed *maxCSInc* rapidly, and soon recovered to use much larger heap sizes.

5.4.2 Comparing with JRockit and HotSpot

JRockit and HotSpot have the ability to adjust the heap size according to a requested throughput or pause time target. However, as we have previously demonstrated, JRockit and HotSpot do not adjust heap size well in response to changing memory allocation. Figure 5.17 compares the throughput of our adaptive collectors with that of JRockit and HotSpot. We carefully choose the initial memory allocation so that the background process imposes the same amount of relative memory pressure as for our adaptive collectors.

However, being an experimental platform, Jikes RVM's compiler did not produce as efficient code as these commercial JVMs. The purpose of our experiments is not to compare

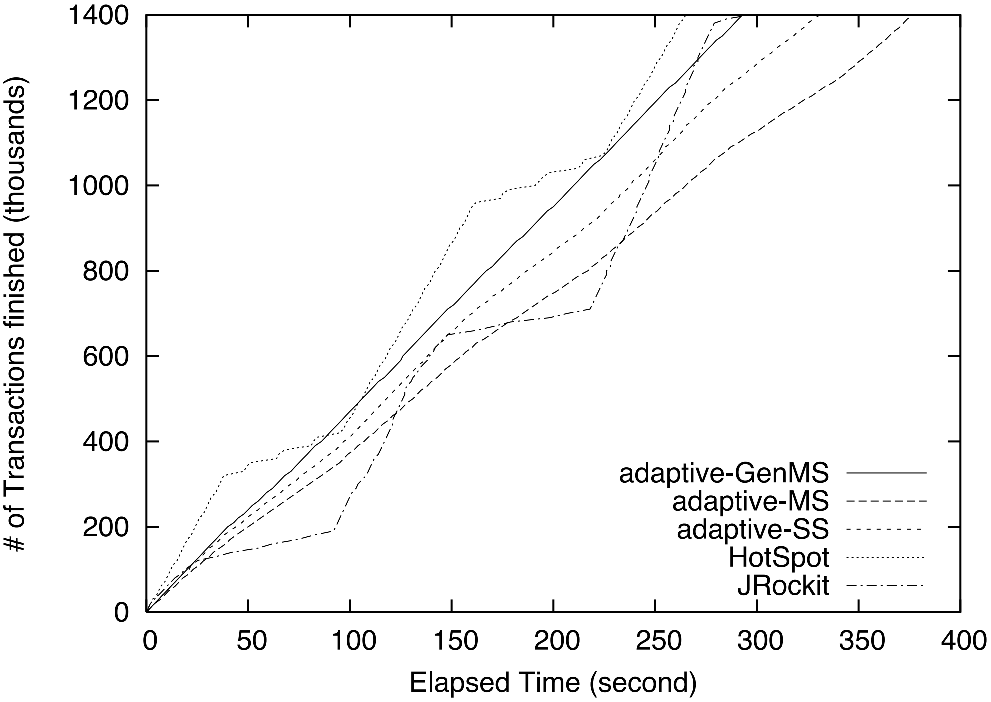


Figure 5.17. Throughput under dynamic memory pressure, versus JRockit and HotSpot.

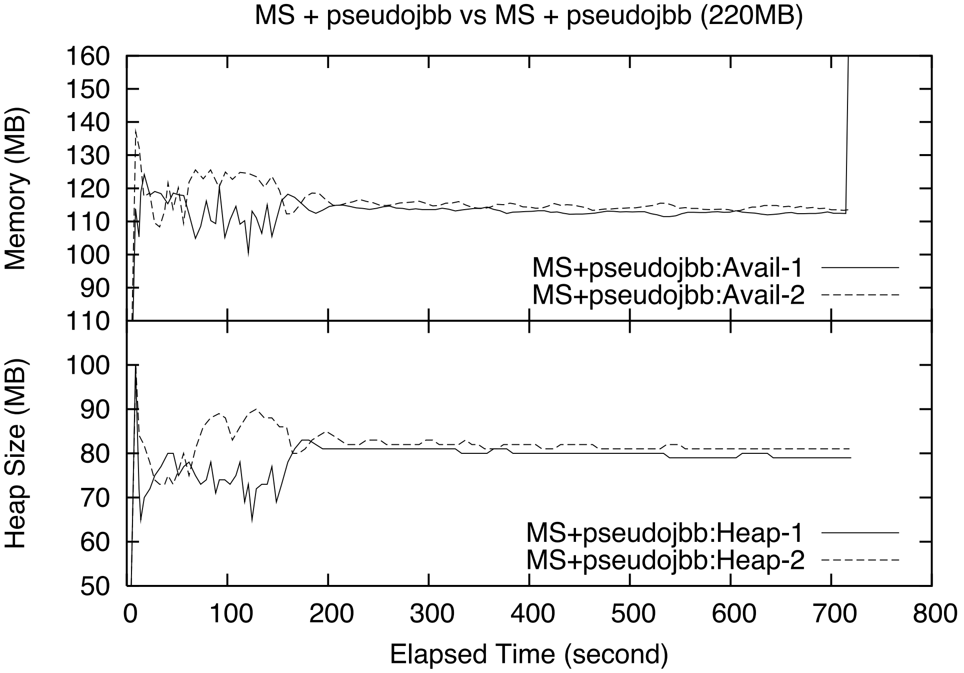


Figure 5.18. Running Two Instances of Adaptive Collectors: Identical Collector and Application

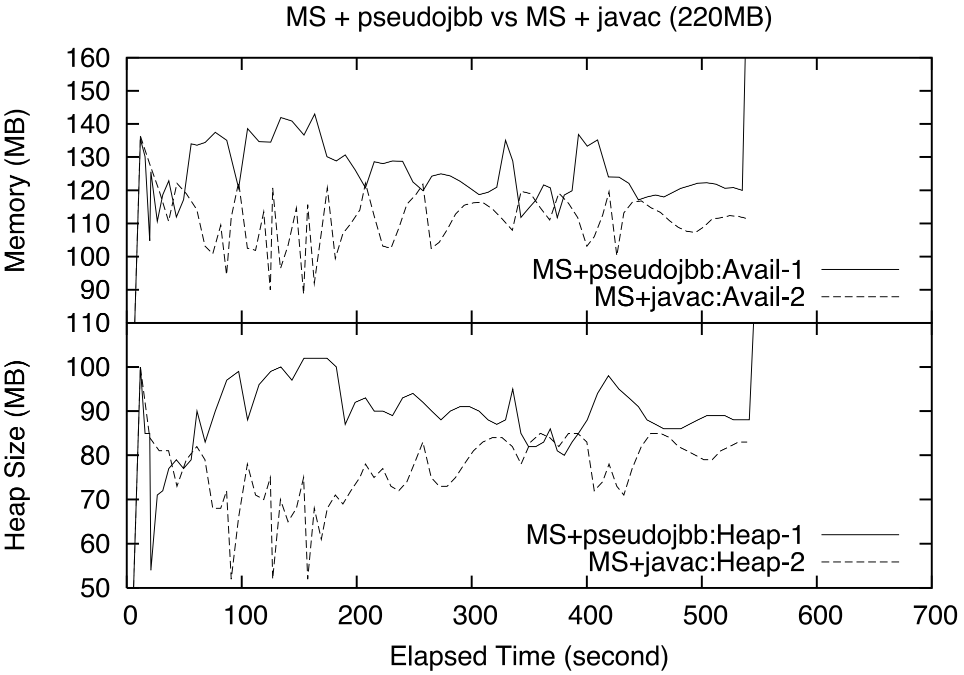


Figure 5.19. Running Two Instances of Adaptive Collectors: Identical Collector and Different Applications

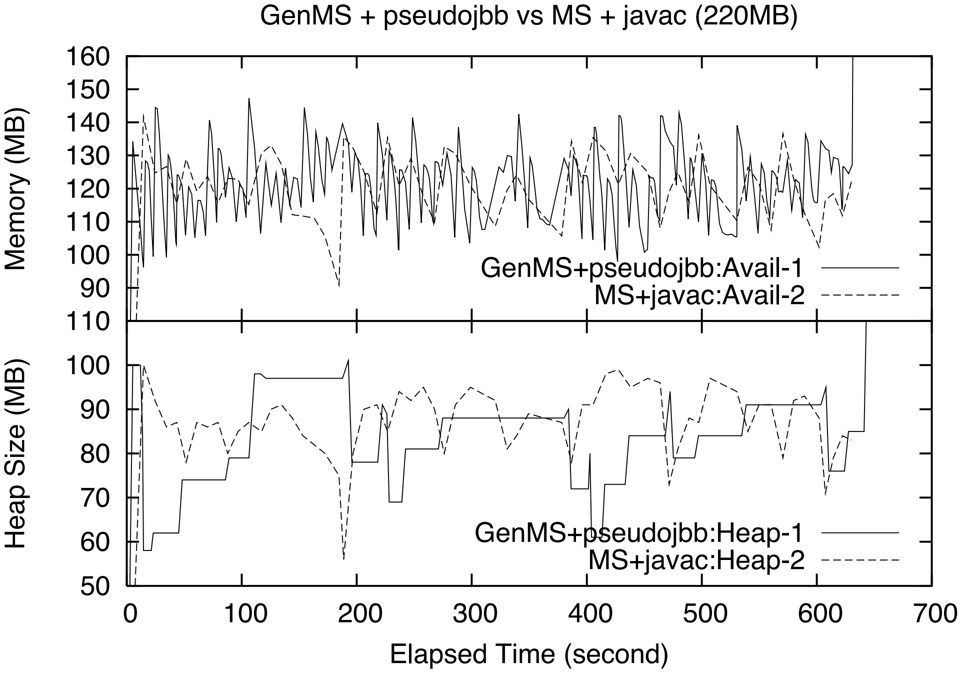


Figure 5.20. Running Two Instances of Adaptive Collectors: Different Collectors and Different Applications

the absolute performance across different JVM implementations, but rather to compare how well they adapts to dynamic memory allocation changes, and how applications are affected. We thus normalize the time for each of them to the total execution time that each JVM takes to run when given ample physical memory (presented as the lower graph in Figure 5.17). The results show that both JRockit and HotSpot experience a large relative performance loss. The flat regions on their throughput curves indicate that they make barely any progress when available memory suddenly shrinks to less than their working set. Meanwhile, our adaptive collector changes the heap size to fit in available memory, maintaining high performance.

5.4.3 Running Multiple JVMs

Finally, we examine how our adaptive collectors interact with each other. We started two instances using adaptive collectors with a certain memory allocation (220MB), and let them adjust their heap sizes independently. We explored several combinations of collector and benchmark: the same collector and benchmark, the same collector and different benchmarks, and different collectors with different benchmarks. The experiments show that, for all these combinations, our adaptive collectors keep CPU utilization at 91% or more. Figures 5.18, 5.19, and 5.20 show the amount of available memory observed by each collector and their adapted heap size over time. We see that, after bouncing around a little, our adaptive collectors tend to converge to heap sizes that give each job a fair share of available memory, even though each works independently. More importantly, they incur only trivial amounts of paging. Note that, in Figure 5.20, the "heap size" curve of GenMS is much smoother than its "memory" curve. This is because the adaptive GenMS collector does not adjust the heap size after small nursery collections. Filtering out small nursery collections prevents the adaptive collector from changing the heap size too aggressively, and thus stablizes the heap heap size.

CHAPTER 6

REDLINE FOR BETTER RESPONSIVENESS

In modern desktop environments, users routinely run highly-graphical user interfaces with resource-intensive applications, ranging from video players and photo editors to web browsers, complete with embedded Javascript and Flash applets. Users now rely on graphical user interfaces to manage and interact with their computer systems. Therefore, maintaining responsiveness is now an important issue, since it has a huge impact on user's experience.

Unfortunately, existing general-purpose operating systems do not provide adequate support for effectively handling these interactive workloads. Current operating systems such as Windows, Linux, and Solaris were designed to use standard resource management policies and algorithms, which in turn were not developed with rapid interactivity in mind. While their CPU schedulers attempt to enhance interactive behavior, their memory managers and I/O managers focus on increasing system throughput rather than reducing latency. This lack of coordination between subsystems, and the fact that they can work at cross purposes, means that pressure on any resource can significantly degrade application responsiveness.

In this part of the dissertation, we focus on how multiple resource managers inside the operating system kernel can be coordinated to maintain system responsiveness under resource contention, producing a system named Redline. We first motivate the work by describing the insufficiency of existing commodity operating systems using a real workload example. We then discuss the characteristics of modern interactive workloads and design considerations for supporting such workloads. Finally, we present an overview of the Redline system, before we present the details of each individual resource manager in later chapters.

6.1 Motivation: Insufficiency of Commodity Operating Systems

Most existing resource managers evolved from designs and implementations developed decades ago when batch-style workloads were prevalent. At that time, their main focus was maximizing the overall throughput, not response time. Despite incremental improvements made over the past years, they are still not able to maintain system responsiveness under heavy resource contention, as we have already shown in Figure 1.1 (Chapter 1).

Their insufficiency is the consequence of two facts. First, these resource managers, designed to achieve better resource utilization but mostly oblivious to response time requirements, are out-dated. While best-effort, priority-based schedulers (or even proportional-share schedulers) are a good match for batch-style applications, they provide limited support for ensuring responsiveness. These schedulers distribute the CPU bandwidth to all running applications in proportion to their priorities (or weights). As a result, they do not provide absolute guarantees on CPU bandwidth allocation and provide no isolation among applications, both of which properties are required by interactive applications. Memory managers reclaim pages based on an LRU policy, making interactive applications vulnerable to memory pressure, since they reference their pages more slowly. Disk I/O schedulers generally sort I/O requests according to physical locations to minimize overall seeks and mechanical delays. Requests from different applications might be mixed in shared queues, regardless of their response time requirements. As a result, disk-intensive applications can easily saturate I/O bandwidth, making applications like a movie player unusable.

More importantly, these resource managers are not coordinated to ensure better responsiveness. They work independently using their own general polices, leading to conflicting decisions in the face of resource contention. As a result, contention on any one of these resources can easily ruin responsiveness. For example, a memory-intensive application can cause the system to evict pages from the graphical user interface, regardless of how hard the CPU scheduler attempts to allocate CPU bandwidth to it, making the system as a whole unresponsive. Similarly, disk-intensive applications can easily saturate I/O bandwidth, making applications like video players unusable. Activities that strain the resources of a system—image or video processing, working with large data files or file systems, or switching frequently between a number of active applications—are likely to cause one of the resource managers to under-allocate resources to some interactive tasks, making those tasks respond poorly.

6.2 Interactive Workloads and Design Considerations

We contend that the reason for the poor interactivity support in commodity operating systems lies in their resource management. We believe that the resource managers of an interactive system have to be coordinated and take into account the characteristics of interactive applications, so that they can manage the resources in an integrated manor towards providing better responsiveness. We now describe the behavior of interactive applications and sketch out several system design requirements based on the study of existing resource management schemes.

6.2.1 The Characteristics of Interactive Workloads

Modern interactive systems have to handle applications whose behaviors are drastically different from those of decades ago, making system resource management more challenging.

- Applications have more complicated workloads that highly depend on their type and external events (e.g., user inputs, interrupts). For example, a service daemon may wake up occasionally in response to certain events, and must consume high CPU bandwidth in a short period of time to process them in a timely manner. The GUI subsystem updates the screen according to mouse operations, introducing wide variance in the amount of work. In general, an application's resource requests tend to be bursty, which may arrive at any time and involve arbitrary amounts of work. Even a single application can change its behavior dramatically. For instance, a movie player normally exhibits periodic behavior. However, if a user attempts to change the video size, play in fast forward, or seek to a new position, it suddenly becomes more CPU intensive or I/O intensive. The combinations of multiple such applications makes the system workload even more difficult to predict.

- A system has to support a large number of latency-sensitive (or response time sensitive) applications. Take Linux as an example. After boot up, a Linux system already has more than 70 processes running, including various kernel threads, service daemons, and the GUI subsystem. These processes are essential to the system. Failure to satisfy these processes can cause poor responsiveness, or potentially even more catastrophic results, such as a system crash if a critical kernel thread does not respond for a period of time. In general, a system has to support hundreds of processes simultaneously, many of which interact with the user either directly or indirectly.

- Applications are highly interdependent. An operation as simple as clicking a mouse involves a surprisingly large number of processes, ranging from kernel threads delivering events and service daemons taking care of inter-process communication, to the GUI subsystem (window manager, desktop manager, Xserver) updating the screen, complete with the application itself performing corresponding actions.

Even though interactive applications are response-time sensitive, they are not as time critical as real-time applications. Moderate delays will not lead to catastrophic failure. Users are willing to tolerate such delays as long as the system makes steady progress and adapts quickly enough. Consequently, the system does not need to adhere to strict guarantees, thus allowing certain flexibility in system resource management.

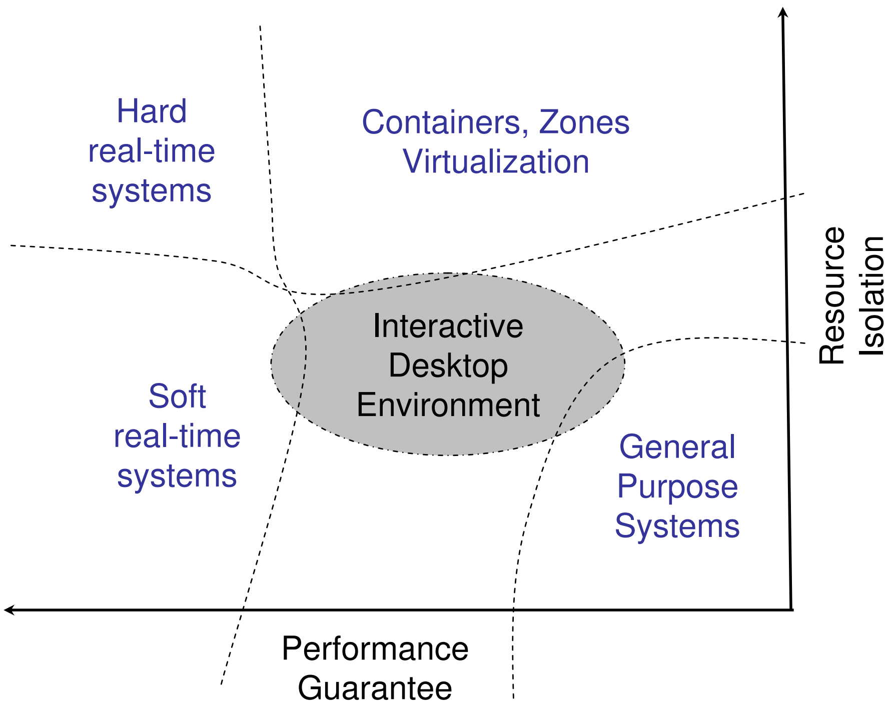


Figure 6.1. Design space for highly interactive systems

In summary, the resource management of an interactive system must be able to support a large number of interdependent response-time sensitive applications, whose workloads are highly dynamic and potentially unpredictable. These applications require certain performance guarantees in order to maintain their responsiveness within a user-acceptable range.

6.2.2 A Taxonomy of System Resource Management

Despite the increasing importance of handling interactive workloads, most system resource management schemes do not match the need of such workloads very well, as Figure 6.1 illustrates. Each of them has its own strengths, but also limitations in terms of handling interactive workloads.

6.2.2.1 General-purpose Systems

As we have previously described, resource managers in general-purpose systems place most of their emphasis on effectively *sharing* resources to maximize the overall throughput, and are not coordinated to handle response time requirements. As a result, there is not enough isolation among applications. The amount of resources allocated to an application can be severely interfered with by other applications in the system. Furthermore, because the manangers usually have no admission control, users can easily overwhelm the system with a massive number of processes. While general-purpose systems have high resource utilization, they have very limited support for response time guarantees.

6.2.2.2 (Soft) Real-time Systems

Real-time systems [81, 74, 60, 96] are designed to provide applications with strict performance guarantees, and thus they have to manage multiple resources for applications in a more integrated manner and take the worse case scenario into account. Every application is required to inform the system of all its resource requirements, by writing a contract or specification [12]. This information is then used to perform admission control, which rejects any request that may potentially cause the violation of a contract to ensure strict performance guarantees. As a consequence of such pessimistic admission control, these systems usually have low resource utilization and are capable of providing guarantees only to a very limited number of applications.

6.2.2.3 Hierarchical Sharing

Several systems manage multiple resources in a hierarchical manner [29, 129, 126]. For example, QLinux [126] manages CPU, disk I/O, and network bandwidth using hierarchical proportional share algorithms. Each level in the hierarchy distributes resources to its lower level, until leaf nodes allocate them to applications. Therefore applications running under different leaf nodes are largely isolated, given that the hierarchy and resource allocation in upper levels does not change. However, applications under the same leaf node can still interfere with each other (depending on the allocation algorithm used), and changing the hierarchy can easily undermine the guarantees.

6.2.2.4 Resource Partitioning

Some systems have the ability to dedicate a portion of their resources into a partition, and isolate the applications running in it from others. For example, the *zones* used by Solaris Containers [90] are configured to act as completely isolated servers, containing their own resources, even their own portion of the file system hierarchy. Virtualization [13] goes even further, allowing each partition to run its own operating system. Such systems provide very strong isolation among different partitions, ensuring that applications in different partitions never interfere with each other. However, due to the dynamic nature and complex interdependence of interactive workloads, it is very difficult to divide applications into disjoint groups and put them into different partitions. Plus, within a single zone or virtual machine, the similar problem of managing multiple interactive applications still exists. Furthermore, it requires a substantial amount of knowledge and effort to configure and maintain such a system, making them not suitable for daily use in interactive environments.

6.2.3 Design Considerations

When extending an existing commodity operating system to provide appropriate support for interactive workloads, there are several important things that have to be considered in the system design.

- Some extra information is required from the user to inform the system as to which applications are interactive, as well as their rough resource requirements. Such information should be concise and simple enough for the user to manage on a daily basis.

- The system should be able to make full use of resources to support as many interactive applications as possible. It needs to achieve high resource utilizataion while avoiding unnecessary over-provisioning of resources. It also has to prevent excessive resource contention from disrupting responsiveness. The fundamental trade-off between *resource utilization* and *performance guarantees* is the key.

- The system should orchestrate various resource managers without undermining their modularity. The interfaces and information propagated among them should be minimal, so that these resource managers can still be reasoned about and managed individually.

- The system should be backward compatible, so that most legacy applications can take advantage of the new feature without changing their source code.

6.3 An Overview of the Redline System

We present Redline, a system that integrates resource management (memory management and disk I/O scheduling) with the CPU scheduler, orchestrating these resource managers to maximize the responsiveness of interactive applications. It works with unaltered interactive applications, providing strong isolation for them while ensuring high system utilization. Figure 6.2 presents an overview of the Redline system, focusing on its support for two types of applications: interactive and best-effort.

1. **Interactive (Iact)**: response-time sensitive tasks that provide services in response to external requests/events. These include not only tasks that interact with users, but also tasks that serve requests from other tasks, such as kernel daemons; and

2. **Best-effort (BE)**: tasks whose performance is not critical from the user's perspective, such as background software updates and virus scanners.

6.3.1 Specification Management

At the top, specification management allows a system administrator to provide specifications for a set of important applications. Redline loads each specification from a file

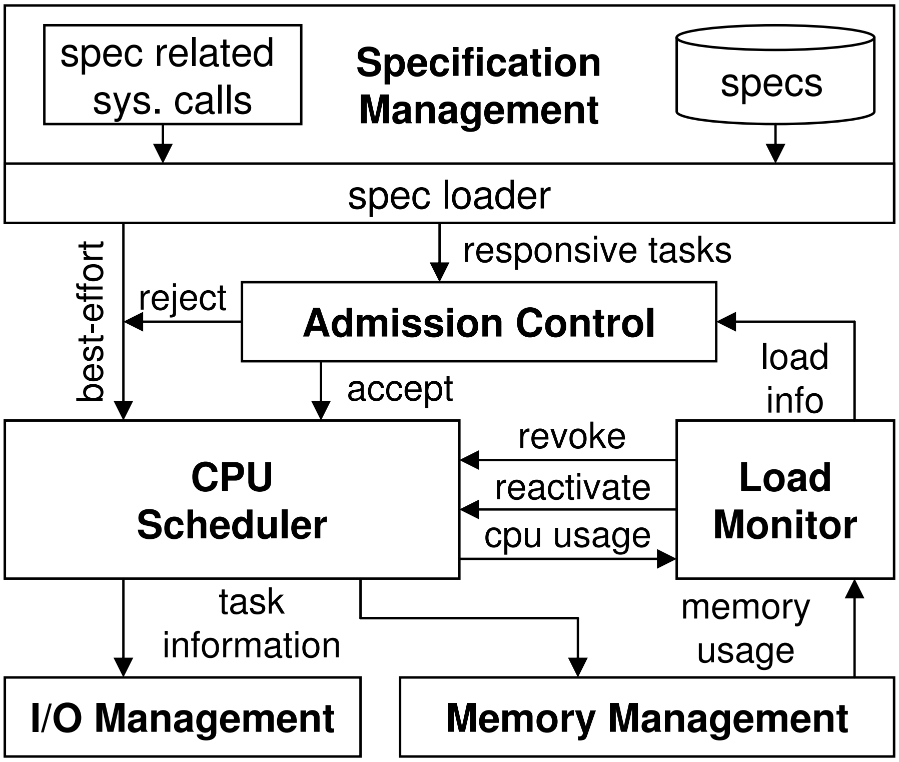


Figure 6.2. The Redline system. Combining integrated resource management with appropriate admission and load control, Redline provides strong isolation for interactive applications while ensuring high system utilization. It maintains system responsiveness, even under heavy resource contention.

whenever the corresponding application is launched. Redline treats any application without a specification as a best-effort task.

6.3.2 Admission and Load Control

Whenever a new task is launched, Redline performs admission control to determine whether the system can accommodate it. Specifically, Redline dynamically tracks the load consumed by the active interactive tasks, and then uses the new task's specification to determine whether enough resources are available to support it as an interactive task.

Unlike real-time systems, which pessimistically reject jobs if the combined specifications would exceed system capacity, Redline optimistically accepts new jobs based on the actual usage of the system, and adapts to overload if necessary. When Redline detects an overload—that the interactive tasks are trying to consume more resources than are available—then the load monitor selects a victim to downgrade, making it a best-effort task. This victim is the task that acts least like an interactive task (i.e., it is the most CPU-intensive). This strategy allows other interactive tasks to continue to meet their response-time requirements. Whenever more resources become available, Redline will promote the downgraded task, again making it interactive.

6.3.3 Integrated Resource Management

Once the admission control mechanism accepts a task, Redline propagates its specification to the memory manager, the disk I/O manager, and the CPU scheduler. The memory manager uses the specification to protect the task's working set, preferentially evicting either non-working-set pages or pages from best-effort tasks. It also maintains a specially controlled pool of free pages for interactive tasks, thus isolating them from best-effort tasks that allocate aggressively. The disk I/O manager assigns higher priorities to interactive tasks, ensuring that I/O requests from interactive tasks finish as quickly as possible. Finally, Redline's extended CPU scheduler provides the required CPU resources for the interactive tasks.

By integrating resource management with appropriate admission and load control, Redline effectively maintains interactive responsiveness even under heavy resource contention. In the following chapters, we will describe Redline's specification management, and then present its CPU, memory, and disk I/O managers. We then evaluate our prototype implementation by stressing the system with various extreme workloads.

CHAPTER 7

REDLINE SPECIFICATIONS

Existing commodity operating systems usually take only a priority parameter (e.g., via nice in Linux and FreeBSD, priority in Windows) for running an application. This parameter is used by the system to infer how resources should be allocated to the application. Usually, higher priority indicates more resource allocation. While this approach works quite well when the system attempts to maximize overall throughput by sharing resources, it is not a good match for handling interactive workloads.

In order to provide better responsiveness, the system boosts the priorities of certain I/O bound processes heuristically in the belief that these are likely to be interactive tasks. However, it becomes almost ineffective in the face of extreme overload. Furthermore, most applications are launched with the same default priority, and ordinary users have very restricted access to use higher priority than that, limiting their ability to protect interactive applications from resource contention.

The fact is that a single parameter is far from adequate to express an application's response time requirements. For the purpose of maintaining responsiveness, the system needs to be informed of at least two pieces of information: *which applications are interactive* and *what are their resource requirements*. We believe users generally have much better information about these than the system itself, since they may have different preferences and requirements under different environments.

Therefore, Redline adopts a different approach, more resembling real-time systems. In Redline, the system administrator selects a set of applications and writes their resource requirements in the form of *lightweight specifications*. These specifications are stored in a file, and loaded automatically by the system when starting an applications (even if the application is launched by an ordinary user). The system applies the specification to the application if it passes the admission test. Unlike real-time systems, Redline does not treat a specification as a strict contract that requires hard performance guarantees. Instead, Redline allows resource over-commitment and addresses overload dynamically to achieve better resource utilization.

In this chapter, we discuss what kind of applications should be considered as interactive in Redline; how to choose specifications for interactive applications; and how Redline manages these specifications.

7.1 Selecting Interactive Applications

Redline is designed to maintain the responsiveness of the system as a whole, rather than to protect just a few individual applications. Therefore, under Redline, the concept of interactive applications covers a wide range. Any application or process that may impact the user's experience, either directly or indirectly, should be considered as potentially interactive:

- Each application that interacts directly with user (e.g., a text editor, a movie player, or a web browser) should have a specification to ensure its responsiveness, depending on the user's preference. However, the application itself depends on other applications and processes that must also be responsive.

- The graphical user interface subsystem (e.g., the X Window server, the window/desktop manager) comprises a set of tasks that are heavily used by most GUI applications, and which interact with users directly. Therefore, these tasks must be given specifications to ensure that the user interface remains responsive.

- Many kernel threads and service daemons are critical to the system. They deliver events, process interrupts, handle inter-process communication, and perform background jobs, such as page laundering and page swapping, most of which are latency sensitive and functionally important. Therefore, they must be considered interactive in Redline to ensure that the system remains stable.

- A range of administrative tools (e.g., bash, top, ls, and kill) should be considered as interactive, so that it is possible for user to diagnose the system and manage the workload whenever necessary, even under extreme load.

Selecting interactive applications can be tricky, due to interdependencies among them. For example, starting bash invokes grep and id several times. If these applications are not marked as interactive, then launching the shell may take an unacceptable amount of time under extreme CPU overloading. Luckily, designating interactive applications need to be done only once. Although we manually select the interactive applications in this work, we expect specifications to be delivered with the system.

7.2 Specification Management

After selecting the set of interactive applications, the system administrator needs to write a lightweight specification for each of them, informing the system of their *estimated* resource requirements, including CPU, memory, and disk I/O. Redline propagates the specification to different resource managers and uses it to coordinate their resource allocations, in an attempt to satisfy as many interactive applications as possible.

7.2.1 Specification Fields

A lightweight specification in Redline is an extension of a *CPU reservation* [74], and most of its fields are self-explanatory. A specification consists of the following fields:

$$\langle \mathit{pathname} : \mathit{type} : C : T : \mathit{flags} : \pi : \mathit{io} \rangle$$

1. *pathname*: the complete pathname to the executable. Redline uses this field to find the matching record for the application at exec() time.

2. *type*: the type of the application. In Redline, it is either interactive (Iact) or best-effort (BE). Redline launches any application without a specification as BE.

3. *CPU reservation* (C:T): These parameters allow an interactive task to reserve C milliseconds of computation time out of every T milliseconds. This reservation is handled by Redline's CPU scheduler (see Chapter 8).

4. *flags*: A set of flags that used by Redline to manipulate specifications. Two important flags include I, which determines whether the specification can be inherited by a child process, and R, which indicates if the specification may be revoked when the system is overloaded.

5. *memory protection period* (π): Redline protects the memory used by interactive applications for a period of time, so that the system does not reclaim their memory too fast simply because of their low memory reference speed (See Chapter 9).

6. *Disk I/O priority* (*io*): the priority that should be used in scheduling disk I/O requests (See Chapter 10). Redline by default gives higher priority to interactive applications, and preferably schedules their request earlier.

To be more concrete, here is an example specification for *mplayer*, an interactive movie player:

$$\langle \text{/usr/bin/mplayer : Iact : 5 : 30 : IR : - : -} \rangle$$

This specification indicates that *mplayer* is an interactive task that reserves 5 ms out of each 30 ms period, whose specification is inheritable, that its interactive status can be revoked if necessary, and whose memory protection period and I/O priority are chosen by Redline automatically.

7.2.2 How to Load Specifications

Redline stores its specifications in a file (/etc/spec/spec.tab). An interactive task either uses the specification loaded from this file when exec() is invoked, or it adopts the

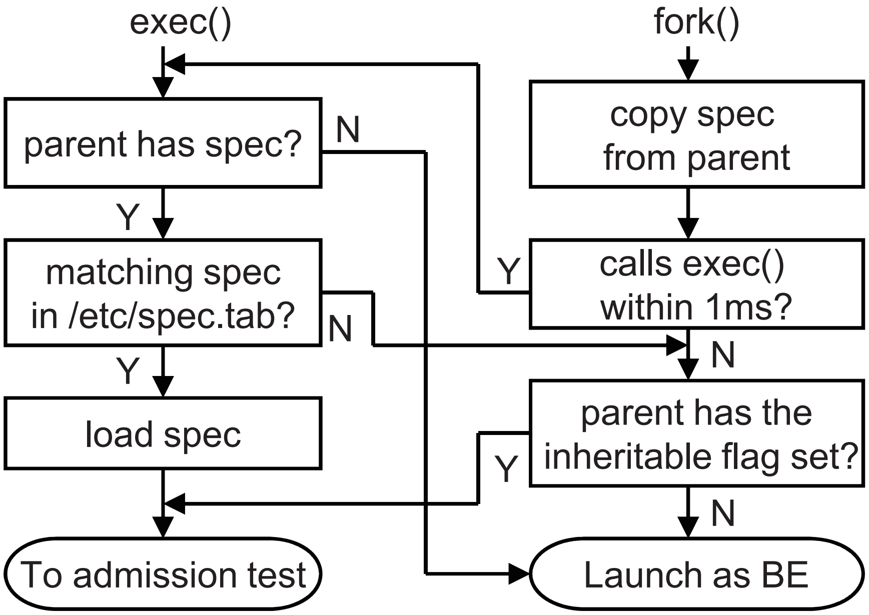


Figure 7.1. Loading specifications in Redline

one inherited from its parent. Figure 7.1 shows how Redline loads the specification for each task.

In practice, most tasks that invoke exec() do so shortly after being forked. Therefore, when a new task is forked, Redline gives it a 1 ms execution window to perform an exec(). If it does so, and if the parent task is itself an interactive task, then Redline searches the specification file for an entry with a matching path name. If a match is found, the specification provided is adopted for this new task.

If there is no entry for the application, or if the task does not invoke exec() during that initial 1 ms window, then Redline examines whether the task should inherit its parent task's specification. If that specification is marked as *inheritable*, then it is applied to the new task. Under all other circumstances, the new task is classified as best-effort.

One thing worth mentioning is that Redline prohibits best-effort tasks from launching interactive tasks, i.e., any task forked by a best-effort task is considered best-effort. The system strictly enforces this rule to prevent users from deliberately overloading the system.

Otherwise, a user can easily push the system beyond the edge by writing a simple program or script that repeatedly executes some applications that have specifications.

If the new task's specification classifies it as interactive, then Redline will submit the task to admission control. If Redline determines that the load is low enough to support the resource needs of this new interactive task, then it is admitted; otherwise, it is demoted to be a best-effort task.

Note that in the absence of any specifications, all tasks become best-effort tasks, and Redline acts like a standard system without special support for interactivity.

7.2.3 How to Choose Specifications

Setting specifications in Redline does not require precise, *a priori* application information. CPU reservations only need to give a rough estimate of the application's CPU requirement, and it is not necessary to consider the worst-case scenario. Redline also attempts to infer other information (i.e., π and *io*) as much as possible and uses those inferences as its default.

We derived specifications for a range of applications by following several simple rules. Because most administrative tools are short-lived, reserving a small percentage of the CPU bandwidth over hundreds of milliseconds is sufficient to ensure responsiveness. While most kernel threads and daemons are not CPU intensive, they tend to be highly response-time sensitive, so their reservation period should be in the tens of milliseconds and a small percentage of the CPU bandwidth should satisfy them. Finally, for interactive applications like a movie player, the reservation period should be around 30 ms to ensure 30 frames per second, which implies that the X server and window/desktop manager should also use the same reservation period. Most of the time, users should be able to select both C and T intuitively based on their preference. Another possibility we have not expolored here would be to provide a simple tool that would allow a user to adjust the reservations, both C and T experimentally, finding the minimal requirements for acceptable performance.

Setting the flags demands more care to ensure that the system behaves normally. For instance, although the command shell (i.e., sh, bash) itself is an interactive application, its specification should not be inherited by any of its children. Otherwise, Redline would treat all applications launched from the command shell as interactive. Furthermore, some applications are so critical to the system's functionality and responsiveness that their specifications should never be revoked, such as init, events, and the X server.

Overall, we found that setting specifications was straightforward. It took the author a single work day to generate a set of specifications manually for about 100 applications in a Linux system using the K Desktop Environment (KDE). Because the specification file is portable, specifications for a wide variety of applications could easily be shipped with an operating system distribution. In an actual deployment, we expect these to be supplied by application developers or system vendors, and the system administrator only needs to choose which of them to enable.

7.3 Discussion

Specification management in Redline is designed to be simple, so that users can manage specifications without deep knowledge. A specification itself is concise and does not require precise application information. Therefore, modest over-specification or under-specification is acceptable, making choosing specifications much easier.

Storing specifications away from applications eliminates any need to modify legacy code, and thus even ordinary users can launch unaltered applications that use the services provided by Redline, making it more usable in practice. Furthermore, Redline acts like a standard system if no specification is provided.

The current implementation is quite naïve. The specification is loaded from a file during exec() and compared for a match, which introduces one extra disk I/O. More importantly, it requires restarting the application once its specification is changed. An alternative approach would use an in-memory hash table loaded during the system startup. We could allow each application to have multiple specifications, each used in different situations, and we could even allow ordinary users to adjust the specifications within a given range, rather than relying totally on the system administrator.

CHAPTER 8

REDLINE: CPU MANAGEMENT

The time-sharing schedulers used by commodity operating systems to manage both interactive and best-effort tasks neither protect against overload nor provide consistent interactive performance. To address these limitations, Redline employs *admission control* to protect it against overload in most cases, and uses *load control* to recover quickly from sudden bursts of activity. Redline also uses an *Earliest Deadline First (EDF)*-based scheduler for interactive tasks to ensure that they receive CPU time as required by their specifications.

8.1 Admission and Load Control

Admission control determines whether a system has sufficient free resources to support a new task. It is required by any system that provides response time guarantees, such as real-time systems, though it is typically absent from commodity operating systems.

In real-time systems, admission control works by conservatively assuming that each task will always consume all of resources indicated in its specification. If the addition of a new task would cause the sum of the specified CPU bandwidths for all active tasks to exceed the available CPU bandwidth, then the new task will be rejected. This conservative approach overestimates the CPU bandwidth actually consumed by aperiodic tasks, which often use much less CPU time than their specifications would indicate. Thus, real-time admission control often rejects tasks that could actually be successfully supported, pessimistically reducing system utilization.

To increase system utilization, Redline uses a more permissive admission control policy. If the CPU bandwidth actually consumed by current interactive tasks is not too high, then it may admit new interactive tasks. Because this approach could lead to an overloaded system if interactive tasks begin to consume more CPU bandwidth, Redline employs load control that allows it to recover quickly from such overload. Consequently, Redline does not provide the inviolable guarantees of a real-time system, but in exchange for allowing short-lived system overloads, Redline ensures far higher utilization than real-time systems could provide.

To maximize utilization while controlling load, Redline strives to keep the CPU bandwidth consumed by interactive tasks within a fixed range. It tracks the actual CPU load, which it uses to drive its admission and load control policies. We describe here how Redline tracks load and how it manages interactive tasks.

8.1.1 Load Tracking

Redline maintains two values that reflect CPU load. The first, R_{load}, represents the actual, recent CPU use. Once per second, Redline measures the CPU bandwidth consumed during that interval by interactive tasks. R_{load} is the exponentially decayed average of these CPU bandwidth samples, and is calculated using the following equations:

$$\begin{aligned} &diff = Load - R_{load} \\ &d = (\tfrac{1+diff}{2}) \times (d_{max} - d_{min}) + d_{min} \\ &R_{load} = (1-d) \times R_{load} + d \times Load \end{aligned}$$

where $d_{min} = 0.05$ and $d_{max} = 0.95$. The rationale behind these equations is to smooth out the load fluctuation and give a conservative estimation. When load increases, the lastest sample (i.e., the higher load) contributes more to the average. When load decreases, the older value contributes more to the average, so that an exceptionally low sample does not lead to significantly under-estimated R_{load}.

Redline keeps the most recent four samples of R_{load} to determine whether the system is overloaded or underloaded to trigger necessary load control. This four-second *observation* *window* is long enough to smooth short-lived bursts of CPU use, and is short enough that longer-lived overloads are quickly detected.

The second value, S_{load}, projects expected CPU load. When an interactive task i is launched, the task's specified CPU bandwidth, $B_i = \frac{C_i}{T_i}$, is added to S_{load}. Since specifications are conservative, S_{load} may overestimate the load. Therefore, over time, S_{load} is exponentially decayed toward R_{load}. Additionally, for short-lived interactive tasks that terminate within the observation window, their contribution to S_{load} is subtracted from that value, thus updating the projection.

8.1.2 Management Policies

Redline uses the most recent R_{load} and S_{load} to conduct its admission control, and the samples of R_{load} in the observation window to determine whether dynamic load control is necessary. Based on these CPU load values, Redline controls interactive tasks through a set of three policies:

8.1.2.1 Admission

When a new interactive task i is submitted, admission control must determine whether accepting the task will force the CPU load above a threshold R_{hi}, thus placing too high a load on the system. If

$$max(R_{load}, S_{load}) + B_i < R_{hi}$$

then i is admitted as an interactive task; otherwise it is placed into the best-effort class. In this way, Redline avoids overloading the system, but does so based on measured system load, thus allowing higher utilization than real-time systems.

8.1.2.2 Revocation

Because of Redline's permissive admission control, it is possible for some interactive tasks to increase their CPU consumption and overload the system. Under these circumstances, Redline revokes the interactive classification of some tasks, demoting them to the best-effort class, thus allowing the remaining interactive tasks to remain responsive.

Specifically, if all the samples in the observation window (i.e. the past four seconds) satisfy

$$R_{load} > R_{max}$$

where $R_{max} > R_{hi}$, then Redline considers the system is overloaded, and revokes tasks until R_{load} falls below R_{hi}. Redline prefers to revoke a task that exhausted its reservation during the observation window, indicating that the task may be more CPU-bound and less interactive. However, if there are no such tasks, Redline revokes the task with the highest CPU bandwidth consumption. Certain tasks are set to be invulnerable to revocation to preserve overall system responsiveness, such as kernel threads and the graphical user interface tasks.

8.1.2.3 Reactivation

When Redline finds that the system has plenty of resources, it will attempt to reactivate a task. Similar to revocation, if all the samples in the observation window satisfy

$$R_{load} < R_{lo}$$

then Redline will reactivate previously-revoked tasks, promoting them from the best-effort class to the interactive class. A task is eligible for reactivation if *both* (a) it passes the usual admission test, *and* (b) its virtual memory size minus its resident size is less than free memory currently available (i.e., it will not immediately induce excessive swapping). We further constrained Redline to reactivate only one task per period of an observation window to avoid reactivating tasks too aggressively.

8.2 The EDF Scheduling Class

Redline extends the Linux system with a new EDF scheduling class in addition to its fair queueing proportional share scheduler, *CFS* [97], and uses it to handle CPU reservations for interactive tasks. Redline's EDF scheduler has its own set of per-CPU run queues just as the

CFS scheduler does. Redline inserts interactive tasks into the run queues of *both* the CFS and EDF schedulers so that each interactive task can receive any unused CPU bandwidth after consuming its reserved bandwidth. The EDF scheduler has precedence over the CFS scheduler. During a context switch, Redline first invokes the EDF scheduler. If the EDF scheduler has no runnable tasks to schedule, then Redline invokes the CFS scheduler.

8.2.1 Scheduling Algorithm

Redline's EDF scheduler allocates CPU cycles to interactive tasks in terms of periods. Each period has a *startTime* and *deadline*, where $deadline - startTime = T$ (the reservation period in its specification). At the begin of each period, an interactive task is entitled to have computation time C, as given in its specification. As a task executes, the EDF scheduler keep track of its CPU usage and deducts the amount consumed. The remaining entitled computation time is kept as *budget*.

Suppose the current time is *now*. An interactive task is *eligible* to use its CPU reservation, only if $startTime \leq now$. Its status can be categorized into the following cases:

- $budget > 0 \wedge deadline > now$: The task still has budget and the deadline is not reached yet. This is the most common case, in which the task can continue to consume its reserved bandwidth.

- $budget \leq 0 \wedge deadline > now$: The task has consumed its entitled reservation before the deadline is reached. It is not eligible for more CPU bandwidth, and is not allowed to perticipate in EDF scheduling until its next period starts.

- $budget > 0 \wedge deadline \leq now$: The task still had budget remaining at the end of a period. It could be that the task does not have enough work or the system is overloaded, making it misses the deadline.

- $budget \leq 0 \wedge deadline \leq now$: The task has missed a deadline, and consumed its entitled reservation after the period ended.

Listing 2 Assign a new reservation period to task *p*

```
 1:    // has budget, deadline not reached
 2: if ($budget > 0$) && ($now < deadline$) then
 3:    return;                          // continue to use the current period
 4: end if
 5:    // has budget, deadline is reached
 6: if ($budget > 0$) && ($now \geq deadline$) then
 7:    if has no interruptible sleep then // there is no voluntary sleep
 8:       return;                       // continue to use the current period
 9:    end if
10: end if
11:
12:    // no budget left: assign a new period
13: dequeue($p$)                             // remove from the runqueue
14: $startTime \leftarrow max(now, deadline)$     // set a new $startTime$
15: $deadline \leftarrow startTime + \mathrm{T}$          // set a new $deadline$
16: $budget \leftarrow max(budget + \mathrm{C}, \mathrm{C})$         // refill the $budget$
17: enqueue($p$)                             // put back into the runqueue
```

If the task has consumed its budget or passes its deadline, the EDF scheduler assigns a new reservation period to the task. The EDF scheduler checks the status of a task and determines whether to assign a new reservation period at the following places using the algorithm in Listing 2.

1. when a new task is initialized,

2. every timer interrupt after a task's budget is updated, and

3. when a task wakes up from sleep.

A task may reach its deadline before expending its budget (see line 6) for the following reasons: it did not actually have enough computation work to exhaust the budget in the past period; it ran into a CPU overload; or it experienced non-discretionary delays, such as page faults or disk I/O. The EDF scheduler differentiates these cases by checking whether the task voluntarily gave up the CPU during the past period (i.e., had at least one interruptible sleep, see line 7). If so, the EDF scheduler considers that it has at least served one CPU request for the task, and assigns a new period to it. Otherwise, it considers that the task missed a deadline and pushes its work through as soon as possible.

If a task consumes its budget before reaching the deadline, it will receive a new reservation period. But the start time of this new period is later than the current time (see line 14). The EDF scheduler considers a task *eligible* for using reserved CPU time only if $startTime \leq now$. Therefore, it will not pick this task for execution until its new reservation period starts. This mechanism prevents an interactive task from consuming more than its entitlement and thereby interfering with other interactive tasks.

As we can see, the EDF scheduling algorithm itself does not guarantee that every task will meet its deadline. It uses only deadlines to determine the execution order. Missing deadlines caused by short bursts of CPU requests do not significantly impact the user experience. However, if the system is indeed overloaded, i.e., too many interactive tasks are exhausting their reservations resulting in many missed deadlines, then the CPU bandwidth consumed by the EDF scheduling class will stay very high for a substantial amount of time. Redline is able to react to such conditions and trigger dynamic load control to resolve the overload.

8.2.2 EDF Scheduler Data Structure and Complexity

At any context switch, the EDF scheduler always picks for execution the eligible task that has the earliest deadline. We implemented its run queue using a tagged red-black tree similar to the binary tree structure proposed in EEVDF [123]. The red-black tree is sorted by the start time of each task. Each node in the tree has a tag recording the earliest deadline in its subtree. The complexity of its enqueue, dequeue, and select operations are all $O(\log n)$, where n is the number of runnable tasks.

The run queue of the Linux CFS scheduler also uses a red-black tree, which is sorted by the virtual time of each task (i.e., the amount of CPU time received divided by the weight of a task). The CFS scheduler always picks the task that has the smallest virtual time for execution. Therefore, Redline's EDF scheduler has the same complexity as the default CFS scheduler, and should have comparable scheduling overhead.

The scheduling overhead mainly consists of two parts: (1) selecting a new task for execution, and (2) maintaining the status of the running task. When the EDF scheduler needs to pick the next task for execution, it first locates the subtree containing the earliest deadline of all eligible tasks. It then traverses this subtree looking for the task with the earliest deadline by following the tag on each node. The CFS scheduler can pick the next task within constant time in most of the cases, if the left-most node of the red-black tree is cached.

Both the EDF and CFS schedulers update the status of the running task once every timer interrupt, and then decide whether to schedule a new one. The Redline EDF scheduler only needs to dequeue/enqueue a task after assigning a new reservation period, while the CFS scheduler has to dequeue/enqueue a task every time its virtual time is updated (i.e., at least once every timer interrupt).

8.3 SMP Load Balancing

Load balancing in Redline is quite simple, because the basic CPU bandwidth needs of an interactive task will be satisfied once it is accepted on a CPU. It is not necessary to move an accepted task unless that CPU is overloaded. The only thing Redline has to do is select a suitable CPU for each new interactive task during calls to `exec()`. Redline always puts a new task on the CPU that has the lowest R_{load} at the time. Once the task passes the admission test, it stays on the same CPU as long as it remains accepted. If the CPU becomes overloaded, Redline will revoke at least one interactive task tied to that CPU. Once turned into best-effort tasks, revoked tasks can be moved to other CPUs by the load balancer. When a revoked task wakes up on a new CPU, Redline will attempt to reactivate its specification if there are adequate resources there.

Redline also prohibits an idle CPU from stealing interactive tasks from other CPUs to avoid unnecessary risk of overloading. For example, suppose CPU1 is 90% busy and CPU2 is 70% busy. If CPU1 steals one interactive task that needs 20% bandwidth from CPU2, then CPU1 will suddenly become overloaded and the task will not be satisfied. Stealing interactive tasks complicates load balancing significantly, but does not offer much benefit. Balancing best-effort tasks to consumed non-reserved bandwidth is good enough for most systems.

Furthermore, in Linux, the load indicator for balancing best-effort tasks is the total weight of all runnable tasks on each CPU. This approach is not appropriate for Redline, since a fraction of CPU bandwidth is dedicated to reservations. For example, a CPU spending 80% of its bandwidth on reservations should support fewer best-effort tasks than a CPU spending 30% on reservations. Therefore, Redline scales the load indicator for each CPU for a fair comparison when balancing best-effort tasks:

$$\sum weight/(1-Rload)$$

8.4 Discussion

Since Redline's admission control takes the current CPU load into account, a user or system administrator can modestly over-specify a task's resource needs. The only danger in over-specification is that a newly-launched task may be rejected by the admission control if the system is sufficiently loaded with other interactive tasks. Once admitted, an interactive task is managed according to its real usage, negating the impact of the over-specification. If an interactive task is under-specified, it will at least make steady progress with the reserved CPU bandwidth allocated to it. Furthermore, if the system is not heavily loaded, an under-specified interactive task will also be allocated some of the remaining CPU bandwidth along with the best-effort tasks. Thus, even under-specified tasks become poorly responsive only if the system load is high.

Redline revokes tasks on a per-thread basis when it detects overload, not on a per-process basis. Therefore, a multi-threaded application may have some of its threads revoked, while others remain interactive. This makes sense for applications that have multiple threads working independently, such as a web browser that uses one thread to handle each tab. However, revoking threads individually can lead to a phenomenon called *priority inversion* where a higher priority task is blocked for a long period of time waiting for a lower priority task. For example, in Redline, an interactive task may wait for a best-effort task to release a lock, but the best-effort task has little chance to run and blocks the interactive task. This problem can be solved by allowing the best-effort task temporarily to borrow the specification from the waiting interactive task [93]. Ideally, the specification could have a flag indicating whether an application should be revoked as a whole or on per-thread basis.

CHAPTER 9

REDLINE: VIRTUAL MEMORY MANAGEMENT

The goal of existing virtual memory managers (VMM) in commodity operating systems is to maximize overall system throughput. Most VMMs employ "use-it-or-lose-it" policies under which memory referencing speed determines allocation: the more pages a task references per second, the larger its main memory allocation.

A task that blocks on I/O is more vulnerable to losing its allocation and then later being forced to page-swap when it awakens. Furthermore, typical VMMs do not isolate each task's allocation, and thus a single memory-intensive task can "steal" the allocations of other tasks that are not actively using their pages, causing the other tasks to page-swap more often. Worse, page swapping itself is the kind of blocking I/O operation that can cause a task to lose more of its allocation, exacerbating the problem.

Because interactive tasks routinely block on user input, they are especially susceptible to allocation loss. If an interactive task has an insufficient allocation, its execution is likely to be interrupted by lengthy page-swap operations, leaving it unresponsive, and making it even more susceptible to further allocation loss. The Redline VMM is designed to keep interactive tasks responsive even after blocking on input. We now describe what modifications Redline has made to the VMM system to achieve this, starting from an overview of its page reclamation mechanism.

9.1 Redline Page Reclamation

In order to keep interactive tasks responsive under memory contention, the VMM must have several important capabilities. First, it has to cache enough pages for interactive tasks,

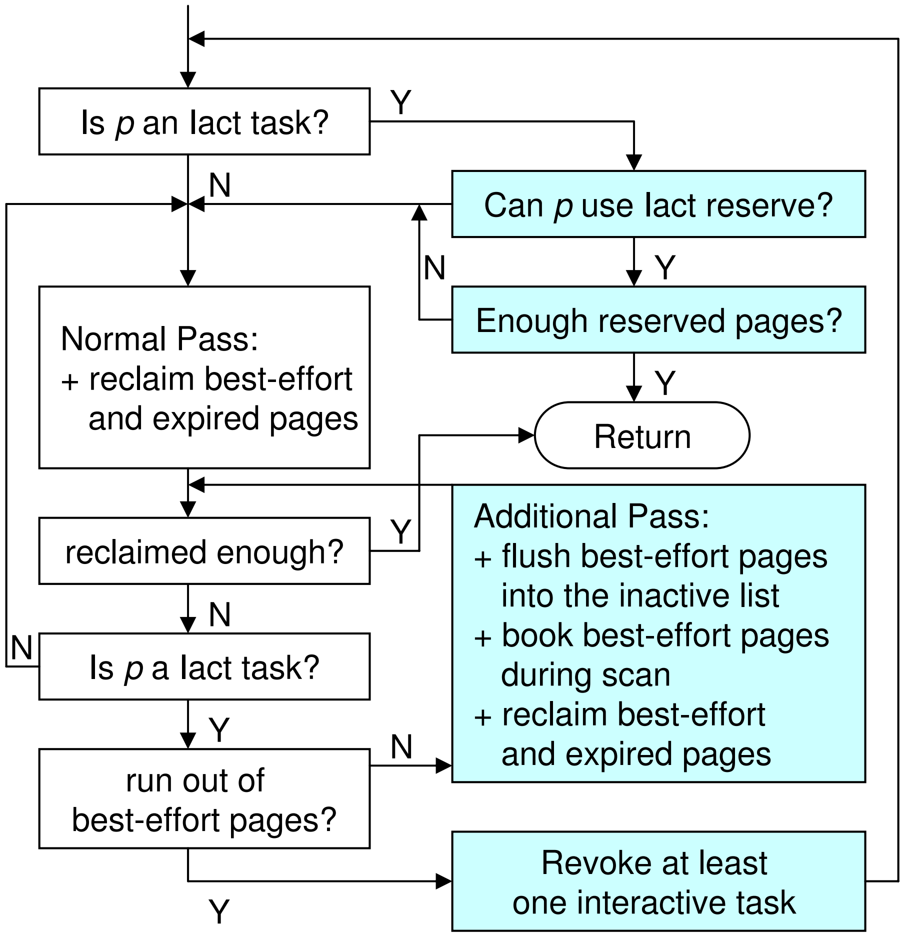


Figure 9.1. Redline VMM page reclamation flow graph. The gray components are additions unique to Redline. Redline protects the working set of interactive applications by skipping unexpired pages; provides limited isolation among them using a rate-controlled reserve; and further reduces the memory reference speed of best-effort application by setting speed-bumps.

so that they do not frequently run into page swapping. It needs to address the conflicts among interactive tasks when not all of them can be satisfied. It has to serve the memory allocation requests coming from interactive tasks as fast as possible, and ensure that best-effort tasks cannot prevent them from getting memory. To achieve these capabilities, Redline extends the standard VMM with several additional mechanisms.

Figure 9.1 gives an overview of Redline's page reclamation algorithm, in which the gray components are additions made by Redline. To be specific, Redline extends the standard

VMM with the following mechanisms to serve the needs of interactive tasks, which we will discuss in detail in the remainder of this chapter:

- **Protecting working sets**: Redline protects the working sets of interactive tasks from being arbitrarily reduced by the VMM, and thus they retain their responsiveness in the face of memory contention. When it cannot cache the working sets of all interactive applications, Redline will dynamically demotes one task to best-effort.
- **Rate-controlled reserve**: Redline uses a small memory reserve to isolate interactive tasks from best-effort tasks, and it further controls the rate of consumption of reserved pages to provide additional limited isolation among interactive tasks.
- **Speed-bump pages**: Redline deliberately slows down the memory reference speed of best-effort tasks under heavy memory pressure to ensure that interactive tasks never lose the battle when competing for pages.

9.2 Protecting working sets

The Linux VMM uses a page replacement algorithm that approximates a *global least recently used (gLRU)* policy. Pages are approximately ordered by recency of use, without consideration of the task to which each page belongs. The VMM cleans and selects for reclamation the least-recently used pages. The implicit goal of this policy is to minimize the total number of page swaps performed by the system, irrespective of how any one process performs.

For an interactive task to remain responsive, a VMM must keep its *working set*—those pages that are currently in active use—resident in memory. Recall the working set definition given by Denning [49]:

> *The working set $W(t,\tau)$ of a process at time t is the collection of pages referenced by the process during the process time interval $(t-\tau,t)$.*

This model indicates that the working set of a process highly depends on the parameter τ. The gLRU policy determines the working set by scanning the list of memory pages. A page is considered not in the working set if it has not been referenced since the last scan, and thus it is subjected to eviction. This means that the interval between two scans of the page list is in fact the working set parameter τ. When memory pressure becomes high, the VMM repeatedly scans through the page list at high frequency, looking for pages to evict, which effectively sets τ to be very small. As a result, it significantly under-estimates the memory needed by those processes that have relatively low memory reference speed.

Consequently, under gLRU a portion of a task's working set may be evicted from main memory if the system-wide demand for main memory is large enough, and if a task does not reference its pages rapidly enough. In the face of such memory pressure, interactive tasks can quickly lose their pages and become non-responsive. In practice, the GUI subsystem usually gets affected first, rather than batch-style background tasks, which is exactly opposite to what users expect and desire.

In Redline, each interactive task can specify a *memory protection period* π. The VMM will evict a page only if it has not been referenced for at least π seconds[1]—that is, if the page has *expired*. By default, $\pi = 30 \times 60$, or 30 minutes if π is not supplied by the specification. This default period requires that a user ignore an interactive application for a substantial period of time before the VMM can evict its working set.

In essence, the Redline VMM attempts to use a fixed working set parameter τ for each interactive task, which is just the memory protection period π. In general, a larger π protects more pages for longer time, behaving closer to locking memory pages, while a smaller π makes the VMM treat a task more like a best-effort one. Instead of using the default, users can choose a suitable π depending on their own needs, and write it in the specification as presented in Chapter 7.

[1] In order to minimize overhead, our Redline implementation actually tracks the last time a page's referenced bit was cleared, rather than tracking the last time the page was referenced.

9.2.1 Page Reclamation Algorithm

The Redline VMM handles the pages of interactive tasks and best-effort tasks differently, as shown in Figure 9.1. If page reclamation is caused by a best-effort task, then the VMM reclaims pages using the system's default VMM mechanism, but with a slight modification: only pages belonging to best-effort tasks and expired pages belonging to interactive tasks may be reclaimed. Among these two types of pages eligible for reclamation, the VMM selects the least recently used pages.

However, if page reclamation is caused by an interactive task, then the VMM first tries to use the *rate-controlled memory reserve* (described in more detail below). If this reserve provides insufficient space, reclamation proceeds as above for best-effort tasks, reclaiming not recently used pages from best-effort tasks and expired pages form interactive tasks. If this attempt at reclamation is also insufficient, the VMM then performs an additional scanning pass which takes all the pages used by best-effort tasks under consideration, and reclaims more recently used pages from them.

9.2.2 Dynamic Load Control

If even the aggressive reclamation performed in the additional scanning pass is insufficient, then there is not enough memory to cache the working sets of all of the interactive tasks. In this case, the Redline VMM demotes some interactive tasks to best-effort tasks. For multi-threaded applications, the VMM will demote the whole process, rather than a portion of its threads. This is because it is unlikely to keep the remaining threads responsive when the working set of the entire process is not cached. After taking this step, the VMM starts the whole page reclamation process over again, attempting to reclaim sufficient main memory space.

Which task to demote is a policy question, whose answer changes depending on system and user preferences. Redline tries to support as many interactive applications simultaneously as possible, therefore the policy it uses is to demote the task that consumes the largest amount of memory so that the remaining ones stay satisfied. Later, Redline will attempt to reactivate the demoted task when memory becomes plentiful.

9.3 Rate-controlled reserve

The Linux VMM forces the allocating task to reclaim pages when the amount of free memory falls below a threshold, which leaves enough memory only for emergency kernel allocations. As a result, a single memory demanding task is able to eat up all of the free memory and to force other tasks to reclaim pages, even if they need to allocate only a small amount of memory. This approach does not provide any isolation among tasks, and therefore it is not desirable for maintaining good response time. Many interactive tasks do repeatedly allocate small memory chunks throughout their lifetime, including the X server, media player, etc. Should such an interactive task fault, it would block during the potentially lengthy reclamation process.

To solve this problem, the Redline VMM maintains a small free-memory reserve (about 8 MB in our implementation) dedicated for interactive tasks, so that they can allocate small memory chunks from it without blocking. Furthermore, the Redline VMM controls the rate at which this reserve is consumed by each interactive task to ensure that it is not exhausted by a single task. This mechanism allows Redline to serve small infrequent memory allocations from interactive tasks much faster, preserving their responsiveness in the face of memory contention. Although an interactive task still may block during reclamation, the reserve makes that situation significantly less likely.

The Redline VMM gives each interactive task a reserve budget b (the default is 256 pages) and records the time t_f when the first page in its budget is consumed. For each page consumed from the reserve, the Redline VMM reduces the budget of the consuming task and then triggers a kernel thread to reclaim pages in background if necessary. We do not want any one task quickly to exhaust the reserve and then affect other tasks, so the rate at which a task consumes reserved pages should not be faster than the rate at which the system can reclaim them. Therefore, the Redline VMM charges each reserved page a cost *c* that is roughly the overhead of one disk access operation (5 ms in Redline). When a task expends its budget, the VMM evaluates the inequality

$$t_f + bc < now$$

where *now* is the current time. If the inequality is true, then the VMM adds *b* to the task's budget. If the inequality is false, the task is consuming reserved pages too quickly. Thus, the VMM prevents the task from using the reserve until *b* pages are reclaimed or the reserve resumes its full capacity.

This is a typical example of the trade-off between resource utilization and isolation that we have to face during the design of Redline. Standard VMMs allow high memory utilization, but fail to provide the isolation needed by interactive tasks. Reserving memory a priori would provide strong isolation, but could lead to significant over-provisioning. The small rate-controlled reserve in Redline offers limited isolation between interactive and best-effort tasks, as well as among interactive tasks themselves, while allowing almost full utilization of the system memory. It serves the needs for the system quite well in practice. We show the effect of this limited isolation in Chapter 11.

9.4 Setting speed-bump pages

A VMM can reclaim only a *clean* (unmodified) page; *dirty* (modified) pages must be copied to the backing store, thus cleaning them, before they can be reclaimed. Sometimes, a best-effort task may dirty pages faster than the VMM can clean them, thus preventing the VMM from reclaiming those pages. The VMM keeps writing its pages back to swap, but makes no actual progress in reclaiming them. If there are too many such unreclaimable pages, the VMM may be unable to cache the working sets of interactive tasks, thus allowing one best-effort task to degrade interactivity.

To prevent best-effort tasks from "locking" memory in this manner, whenever the Redline VMM is aggressively searching for pages to reclaim (in the additional scanning pass shown in Figure 9.1) and it finds a page belonging to a best-effort task, it removes access permissions to that page and marks it as a *speed-bump page*. If the task then references that page, execution traps into the kernel and the VMM notices that the referenced page is a speed bump. Before restoring access permissions and resuming execution, Redline suspends the task briefly (e.g., 100 ms). The VMM therefore slows the task's memory reference rate, giving the VMM enough time to reclaim more of its pages.

Redline sets speed-bump pages only when it fails to make reasonable progress while there are still plenty of pages used by best-effort tasks. Furthermore, it does not need to set many speed bump pages. Usually a few speed-bumps (32 pages in our implementation) allow the VMM to reclaim enough pages. As a consequence, best-effort tasks are forced to take page faults repeatedly in their later execution, making their memory reference speed even lower. Combining this mechanism with working set protection, Redline ensures that interactive tasks never lose the battle when competing for memory pages.

9.5 Discussion

For any task to be responsive, the operating system must cache its working set in main memory. Many real-time systems conservatively "pin" all pages of a task, preventing their removal from main memory to ensure that the working set is cached. In contrast, Redline protects any page used by an interactive task within the last π seconds, thus protecting the task's working set while allowing its inactive pages to be evicted. In choosing a value for π, it is important not to make it too small, causing the system to be behave like a standard, commodity OS. It is safer to overestimate π, although doing so makes Redline behave more like a conservative real-time system. This method of identifying the working set works well under many circumstances with reasonable choices of π.

However, there are other mechanisms that estimate the working set more accurately. By using such mechanisms, Redline could avoid the dangers caused by setting π poorly. One such alternative is the working set measurement used in CRAMM [134]. CRAMM maintains reference distribution histograms to track each task's working set on-line with low overhead and high accuracy. While we believe that this approach is likely to work well, it does not guarantee that the working set is always properly identified, given the seemingly random behavior of interactive applications. Specifically, if a task performs a *phase change*, altering its reference behavior suddenly and significantly, this mechanism will require a moderate period of time to recognize the change. During this period, CRAMM could substantially under- or over-estimate application working set sizes. However, we believe such behavior is likely to be brief and tolerable in the vast majority of cases.

CHAPTER 10

REDLINE: DISK I/O MANAGEMENT

The disk I/O management subsystem of existing operating systems is quit complicated. Typically, an I/O request has to go through several layers in the system before its data can be actually read from or written to a disk. Each of these layers takes care of part of the job:

- **Caching**: The system usually caches disk files in memory to avoid unnecessary I/O operations, in what is referred to as a *page cache* in Linux. The page cache can also buffer some file updates and later flush them to disk en masse to achieve higher throughput.

- **File system and journaling**: The file system organizes the files on disk and determines the allocation of their data. Journaling ensures that the updates are written to disk in a consistent manner, so that the file system can be restored after a crash.

- **Block device layer**: This layer first transforms application's I/O requests to a low-level disk representation, and then determines in which order they should be dispatched to the device driver to carry out actual I/O operations.

Like the VMM, these components of the I/O management of a general-purpose operating system do not distinguish between interactive and best-effort tasks. The policies that determine when and in what order pages are read from and written to disk are designed to optimize system throughput and are oblivious to CPU scheduler goals. This obliviousness can lead the I/O manager to schedule the requests for best-effort tasks before those of interactive tasks in a way that substantially increases response times. Similarly, these components work independently without any coordination, and thus, negative interactions among them can easily disrupt responsiveness.

In this chapter we present the problems caused by isolation of these components and describe how the Redline I/O manager coordinates these components so that interactive tasks can finish their I/O operations as quickly as possible. We present necessary background knowledge to help understand both the problems we identify as well as their solutions. Although the discussion in this chapter is specifically tied to the current Linux implementation, the general principles should apply to many general purpose operating systems, where similar problems exist.

10.1 Page Cache and Journaling

Normally, Linux buffers data from write operations in its page cache. This is called *buffered I/O*. The page cache is allowed to cache a certain number of dirty file pages, which is controlled by a single threshold. When this threshold is reached, any task that attempts to perform write operations has to launder enough dirty pages before it can put its own data in the page cache. As a result, a single best-effort task can pollute the page cache with enough dirty pages so that interactive tasks have to perform lengthy page laundering when they want to write, reducing their responsiveness. However, compared to this, things can get much worse when file system journaling is involved.

10.1.1 Compound Transactions in Journaling

The *ext3* journaling file system is the default for most Linux distributions. Like many journaling file systems, *ext3* commits its updates as atomic transactions, each of which writes a group of cached, dirty pages along with their new metadata. Its implementation is designed to maximize system-wide throughput, sometimes to the detriment of CPU scheduling goals. We describe here a particular problem with this file system's implementation that Redline fixes. Although this particular problem is specific to Linux's *ext3*, it is representative of the way in which any operating system component that manages system resources can undermine interactivity.

Consider two tasks: an interactive task P_i, and a best-effort task P_{be}, which simultaneously use the write() system call to save data to some file on the same *ext3* file system. These system calls will not immediately initiate disk activity. Instead, the data written via this mechanism will be buffered as a set of dirty pages in the file system cache. Critically, these pages will also be added to a single, global, *compound* transaction by *ext3*. This transaction will thus contain dirty pages from any file written by any task, including pages written by both P_i and P_{be}.

Consider the case that P_{be} writes a large amount of data through write(), while P_i writes a small amount. Furthermore, after both tasks have performed these write() operations, suppose that P_i performs an fsync() system call to ensure that its updates are committed to disk. Because of the compound transactions used by *ext3*, P_i will block until both its own dirty pages *and* those of P_{be} are written to disk.

If the system caches too many pages written by P_{be}, then the fsync() operation will force P_i to become noticeably unresponsive. This poor interaction between compound transactions and fsync() occurs not only for *ext3*, but also for *Reiser FS* [107]. Under Linux, the *dirty threshold* d is a system-wide parameter that determines what percentage of main memory may hold dirty pages—pages that may belong to any task—before a disk transfer is initiated to "clean" those pages. By default, $d = 10\%$, making it possible on a typical system for 100 – 200 MB of dirty pages to be cached and then written synchronously when fsync() is called. Worse, during committing the journaling process has to compete for I/O bandwidth with other processes that are laundering dirty pages, making P_i wait even longer.

Redline takes a number of steps to limit these effects. First, Redline assigns different dirty thresholds for each type of task (RT:10%, Iact:5%, BE:2MB), which relieves interactive tasks from performing unnecessary page laundering. Redline further restricts the threshold for best-effort tasks to a constant limit of 2MB, ensuring that no matter the size of main memory, best-effort tasks cannot fill the compound transactions of some journaling file system with a large number of dirty pages. Finally, Redline assigns the kernel task that manages write operations for each file system (in Linux, kjournald) to be an interactive task, ensuring that time-critical transaction operations are not delayed by other demands on the system.

10.1.2 Large Direct I/O Operations

In addition to buffered I/O, most operating systems also support direct I/O, which allows a task to manage its own buffer for file data. However, although direct I/O bypasses the page cache, it can still cause interactive tasks to lose their responsiveness when interacting with journaling.

To understand how this can happen, we need to go a little deeper into the compound transaction. The life-time of a transaction in a journaling file system usually consists of four stages:

- Running: the transaction is open and accepting incoming I/O requests.
- Locked: the transaction is preparing for commit and is waiting for all ongoing operations to finish. During this period of time, it no longer accepts new I/O operations, and thus all incoming operations are suspended. The system will not create a new transaction at this point.
- Committing: the transaction is fully closed and its content is under write-back. As soon as the previous transaction enters this stage, the system can create a new transaction to serve the suspended, as well as newly arrived requests, while committing the previous one.
- Committed: the data and metadata of the transaction are written back to the disk, and the commit is finished.

The journaling system effectively freezes all the I/O operations as long as the current transaction stays Locked. Therefore, if the current transaction is held in the Locked stage for too long, it can significantly degrade the responsiveness of interactive tasks. Unfortunately, this can indeed happen in the current Linux implementation. Linux treats direct I/O to an *existing* file as a single operation, while buffered I/O is automatically broken down to page granularity. As a result, one direct I/O with a huge buffer can cause the system to fail to respond to I/O requests for a long period of time.

Redline makes a small change in the file system to solve this problem. Every time the file system locates the next data block, it also checks whether the current transaction is in the Locked stage. If that is the case, it immediately closes the current operation and creates another operation for handling the remaining part of the direct I/O request. In this way, Redline effectively splits the direct I/O operation to smaller pieces so that the current transaction can quickly enter the committing stage and creates an new transaction to process later requests.

10.2 Block device layer and I/O scheduler

The Linux block device layer is designed to be extensible, allowing each block device to have its own I/O scheduler, which is refered to as an *elevator*, because managing disk seeks is like scheduling elevator movement. Figure 10.1 presents the structure of the Linux block device layer. Every I/O operation has to pass congestion control before it can be accepted (the left part). Accepted requests are stored in a *request queue*, which is managed by a specific elevator (the middle part). In our case, Figure 10.1 shows the default elevator, dubbed *Completely Fair Queuing (CFQ)*. The I/O scheduler determines when and in which order requests are dispatched to its corresponding device driver. Finally, the device driver serves dispatched requests using the SCAN (elevator) algorithm.

Much like the journaling file systems described above, a block device layer has unified data structures, thresholds, and policies that are applied irrespective of the tasks involved.

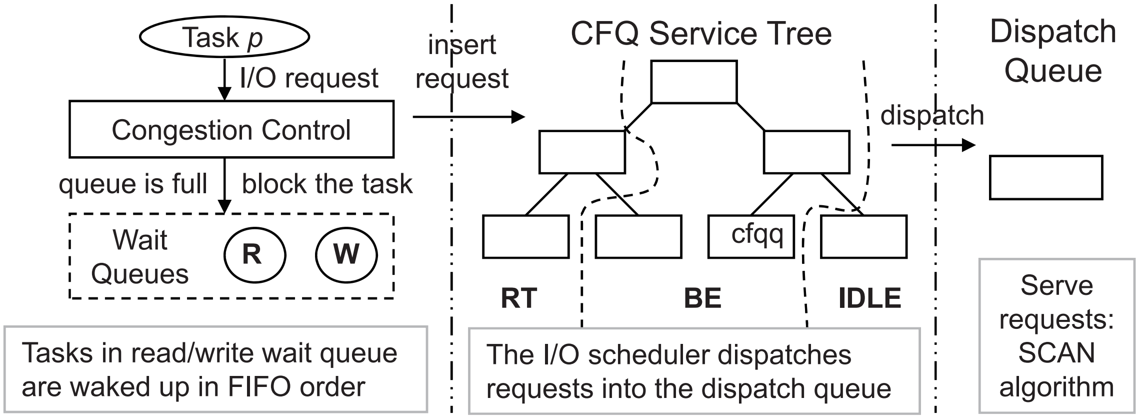

Task p
I/O request
Congestion Control
queue is full
block the task
Wait Queues
R
W
Tasks in read/write wait queue are waked up in FIFO order
insert request
CFQ Service Tree
cfqq
RT
BE
IDLE
The I/O scheduler dispatches requests into the dispatch queue
dispatch
Dispatch Queue
Serve requests: SCAN algorithm

Figure 10.1. How I/O requests are organized and processed in block device layer in Linux.

These components are typically designed to maximize system-wide throughput. However, the unification performed by these components may harm the responsiveness of interactive tasks. Redline addresses these problems by handling these components of the block device layer on a per-task-type basis.

10.2.1 Unified Threshold for Congestion Control

Although the request queue of CFQ internally organizes I/O requests into classes (real-time, best-effort, and "idle"), it has a maximum capacity (128 for read and write each) that is oblivious to these classes. When a new request is submitted, the *congestion control mechanism* examines only whether the request queue is full, with no consideration of the requesting task's class. If the queue is full, the submitting task blocks and is placed in a FIFO-ordered wait-queue. Thus, a best-effort task might rapidly submit a large number of requests, thus congesting the block device and repeatedly delaying interactive tasks that are performing I/O operations.

Redline addresses this problem by using multiple request queues, one per task class. If one of the queues fills, the congestion control mechanism will block processes only in the class associated with that queue. Therefore, no matter how many requests have been generated by best-effort tasks, those requests alone cannot cause an interactive task to block.

10.2.2 Shared Queues for I/O Requests

In addition, while the default request queue for CFQ is capable of differentiating between various request types, it does not provide sufficient isolation for interactive tasks. Specifically, once a request has been accepted into the request queue, it awaits selection by the I/O scheduler to be placed in the dispatch queue, where it is scheduled by a typical elevator algorithm to be performed by the disk itself. The default CFQ scheduler not only gives preference to requests based on their class, but also respects the priorities that tasks assign requests within each class. However, each buffered write request—the most common kind—is placed into a set of shared queues in the best-effort class irrespective of the task that submitted the request. Therefore, best-effort tasks may still interfere with the buffered write requests of interactive tasks by submitting large numbers of buffered write requests.

Redline adds an interactive class Iact to request queue management, matching its CPU scheduling classes. All write requests are placed into the appropriate request queue class based on the type of the submitting task. The I/O scheduler prefers requests from the interactive class over those in the BE class, thus ensuring isolation of the requests of interactive tasks from BE tasks.

Additionally, the specification for a task (see Chapter 7) includes the ability to specify the priority of the task's I/O requests. If the specification does not explicitly provide this information, then Redline automatically assigns a higher priority to tasks with smaller T values. Redline organizes requests on a per-task basis within the given class, allowing the I/O scheduler to provide some isolation between interactive tasks.

Finally, CFQ by default does not guard against starvation. A task that submits a low-priority I/O request into one of the lower classes may never have that request serviced.

Redline modifies the I/O scheduler to ensure that all requests are eventually served, preventing starvation. Specifically, Redline records the time when each request is accepted into the request queue and puts them into an additional FIFO list. At the end of each dispatch, the I/O scheduler also checks the first request of this list to see whether it has been there too long. If so, Redline will schedule the request.

10.3 Discussion

Most interactive applications perform a relatively small amount of disk I/O in an infrequent manner, which usually does not exceed the system capacity. We believe that a preemptive priority-driven approach combined with per-process I/O request management will work well enough for commodity operating systems and will minimize modification of the kernel. However, many real time systems have their own specialized disk I/O schedulers [81, 126, 60]. These I/O schedulers offer more precise bandwidth control and could potentially be used by Redline.

CHAPTER 11

REDLINE: EXPERIMENTAL EVALUATION

We have implemented the Redline system as an extension to the Linux kernel. We have also generated a set of specifications for over 100 applications for a Linux system that uses the K Desktop Environment (KDE). Recall that Figure 1.2 showed that the execution of a single compilation command (`make -j32`) significantly degrades responsiveness on a standard system, while Redline is able to keep the whole system responsive, as well as play the video smoothly. In this chapter, we further evaluate the effectiveness of our Redline implementation by stressing the system with a variety of extreme workloads, including fork “bombs”, memory “bombs”, and bursty, large disk I/O requests.

11.1 Methodology Overview

In this section, we present a methodology overview of how we evaluate our Redline implementation.

11.1.1 Platform

We perform all measurements on a system with a 3 GHz Pentium 4 CPU, 1 GB of RAM, a 40GB FUJITSU 5400RPM ATA disk, and an Intel 82865G integrated graphics card. The processor employs *simultaneous multi-threading (SMT)* (i.e., Intel’s HyperThreading), thus appearing to the system as two processors. The processor has a 12 KB L1 instruction and an 8 KB L1 data cache, and an unified 512 KB L2 cache.

We use a Linux kernel (version 2.6.22.5) patched with the CFS scheduler (version 20.3) as our control. Redline is implemented as a patch to this same Linux version. For all experiments, the screen resolution was set to 1600 x 1200 pixels. All experiments used both of the SMT-based virtual CPUs except when measuring the context switch overhead. Furthermore, we ran each experiment 30 times, taking both the arithmetic mean and the standard deviation of all timing measurements.

11.1.2 Application Settings and Inputs

In order to keep the whole system responsive, Redline considers many applications to be interactive. Table 11.1 shows only a subset of the task specifications that are used in our experiments when evaluating Redline. It includes the *init* process, *kjournald*, the X11 server *Xorg*, KDE's desktop/window manager, the *bash* shell, and several typical interactive applications. We left the memory protection period (π) and I/O priority empty in all the specifications, letting Redline choose them automatically.

	C:T (ms)		C:T (ms)
init	2:50	kjournald	10:100
Xorg	15:30	kdeinit	2:30
kwin	3:30	kdesktop	3:30
bash	5:100	vim	5:100
mplayer	5:30	firefox	6:30

Table 11.1. A subset of the specifications used in the Redline experiments. The memory protection period and I/O priority are chosen automatically by Redline.

The movie player, *mplayer*, plays a 924.3 Kb/s AVI-format video at 25 frames per second (f/s) with a resolution of 520 x 274. To give the standard Linux system the greatest opportunity to support these interactive tasks, we set *mplayer*, *firefox*, and *vim* to have a CPU scheduler priority of -20—the highest priority possible. Also note that a pessimistic admission test, like that of a real-time system, would not accept all of the applications in Table 11.1, because they would overcommit the system. Redline, however, accepts these as well as many other interactive tasks.

11.2 CPU Scheduling

Redline extends the Linux CPU scheduler with an EDF scheduling class to serve the needs of interactive application. For the Redline system to be practical, it CPU scheduler must have competitive performance with respect to that of Linux (i.e., the CFS scheduler). Therefore, we first measures the overhead of Redline's EDF scheduler and then evaluate its effectiveness.

11.2.1 EDF Scheduler Overhead:

We compare the performance of the Redline EDF scheduler with the Linux CFS scheduler. Figure 11.1(a) presents the context switch overhead for each scheduler as reported by *lmbench*. From 2 to 96 processes, the context switch time for both schedulers is nearly indistinguishable.

However, when lmbench measures context switching time, it creates a workload in which exactly one task is unblocked and ready to run at any given moment. This characteristic of lmbench does not actually exercise the data structure for these schedulers enough, so we further compare these schedulers by running multiple busy-looping tasks, all of which are ready to run at any moment. We tested workloads of 1, 20, 200, and 2000 tasks. We perform this test twice for Redline, launching best-effort tasks to test its CFS scheduler, and launching interactive tasks to test its EDF scheduler. In the latter case, we assigned specification values for C:T as 1:3, 1:30, 1:300 and 1:3000 respectively, thus forcing the Redline EDF scheduler to perform a context switch almost every millisecond. Note that with these specification values, the Redline EDF scheduler is invoked more frequently than a CFS scheduler would be, running roughly once per millisecond (whereas the usual Linux CFS quantum is 3 ms).

The number of loops is chosen so that each run takes approximately 1400 seconds. Figure 11.1(b) shows the mean total execution time of running these groups of tasks with Linux CFS, Redline CFS, and Redline EDF. Note that the y-axis starts from 1300 seconds

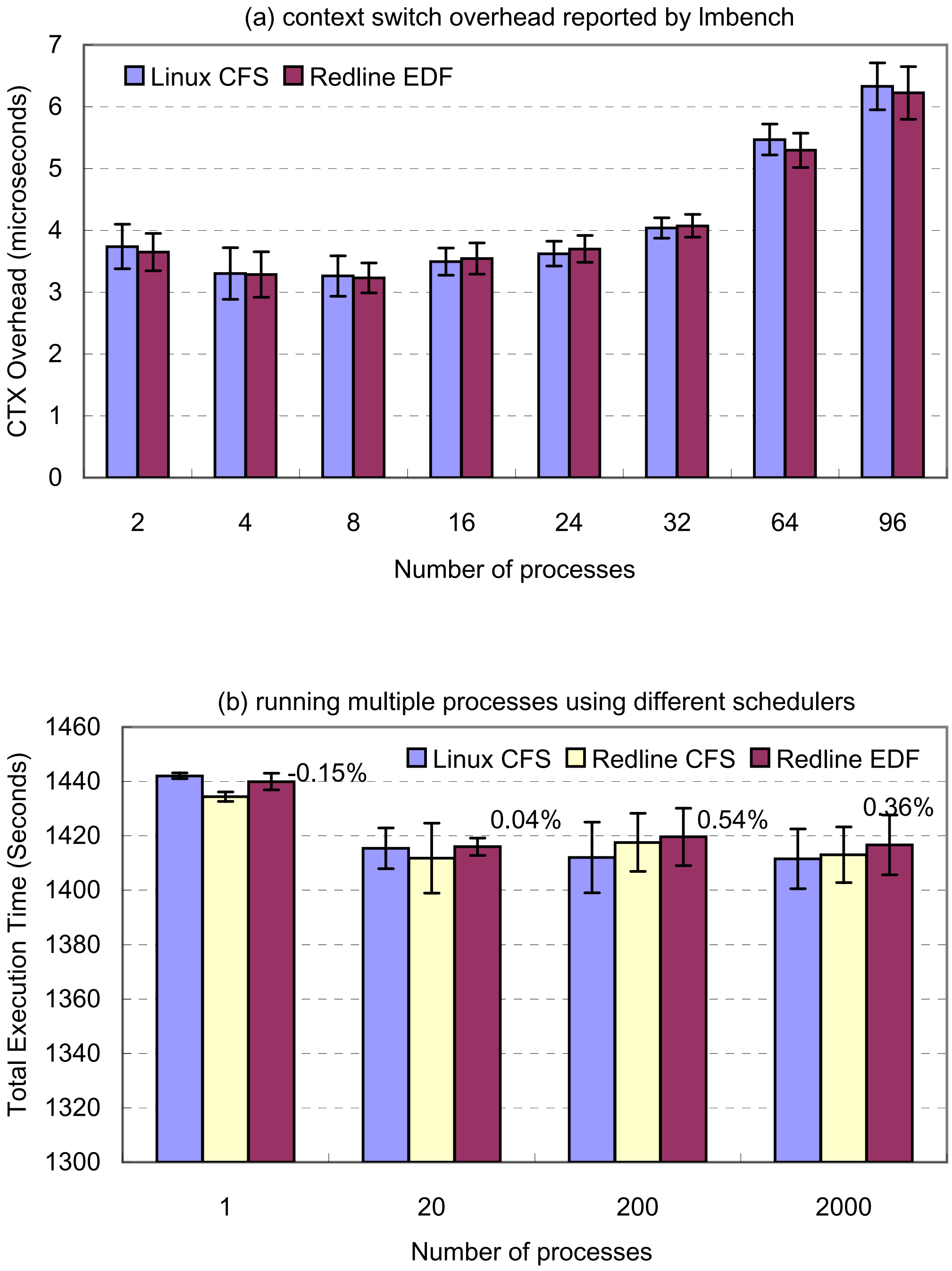


Figure 11.1. An evaluation of CPU scheduling overhead. Figure (a) shows the context switching times as evaluated by *lmbench*. Figure (b) shows the total running time of varying numbers of CPU intensive tasks. Note the y-axis starts at 1300 to make the minor variation visible.

to make the differences visible, because they are so small. As shown in this graph, in the worst case the Redline EDF scheduler adds 0.54% to the running time, even when context switching far more frequently than the CFS schedulers. Also the CFS scheduler in Redline has almost the same performance compared to its counterparts in Linux. Note that the total execution time of these experiments was bimodal and, as shown by the error bars, made the variance in running times larger than the difference between results. The overhead of using the Redline schedulers is negligible.

11.2.2 CPU Overload – Fork Bombs

We now evaluate the ability of Redline to maintain the responsiveness of interactive tasks. First, we launch mplayer as an interactive task, letting it run for a few seconds. Then, we simultaneously launch many CPU-intensive tasks, thus performing a fork bomb. Specifically, one task forks a fixed number of child tasks, each of which executes an infinite loop. The parent then kills the child tasks after 30 seconds. We performed two tests for Redline: the first runs the fork bomb tasks as best-effort, and the second runs them as interactive. In the latter case, the fork bomb tasks were given CPU bandwidth specifications of 10:100. For the Linux test, the interactive task had the highest possible priority (-20), while the fork bomb tasks were assigned the default priority (0).

Figure 11.2 shows the frame rate of achieved by mplayer over time during these tests. In Figure 11.2(a), the fork bomb comprises 50 tasks, while Figure 11.2(b) shows a 2000-task fork bomb. Both of these figures show, for Redline, best-effort and interactive fork bombs. Under Linux with 50 tasks, mplayer begins normally. After approximately 10 seconds, the fork bomb begins and mplayer has to share the CPU with 50 more tasks, receiving less than it needs. Since at that time, it is able to play at only about 10 frames per second, the user has to watch the video in slow motion. Amusingly, after the fork bomb

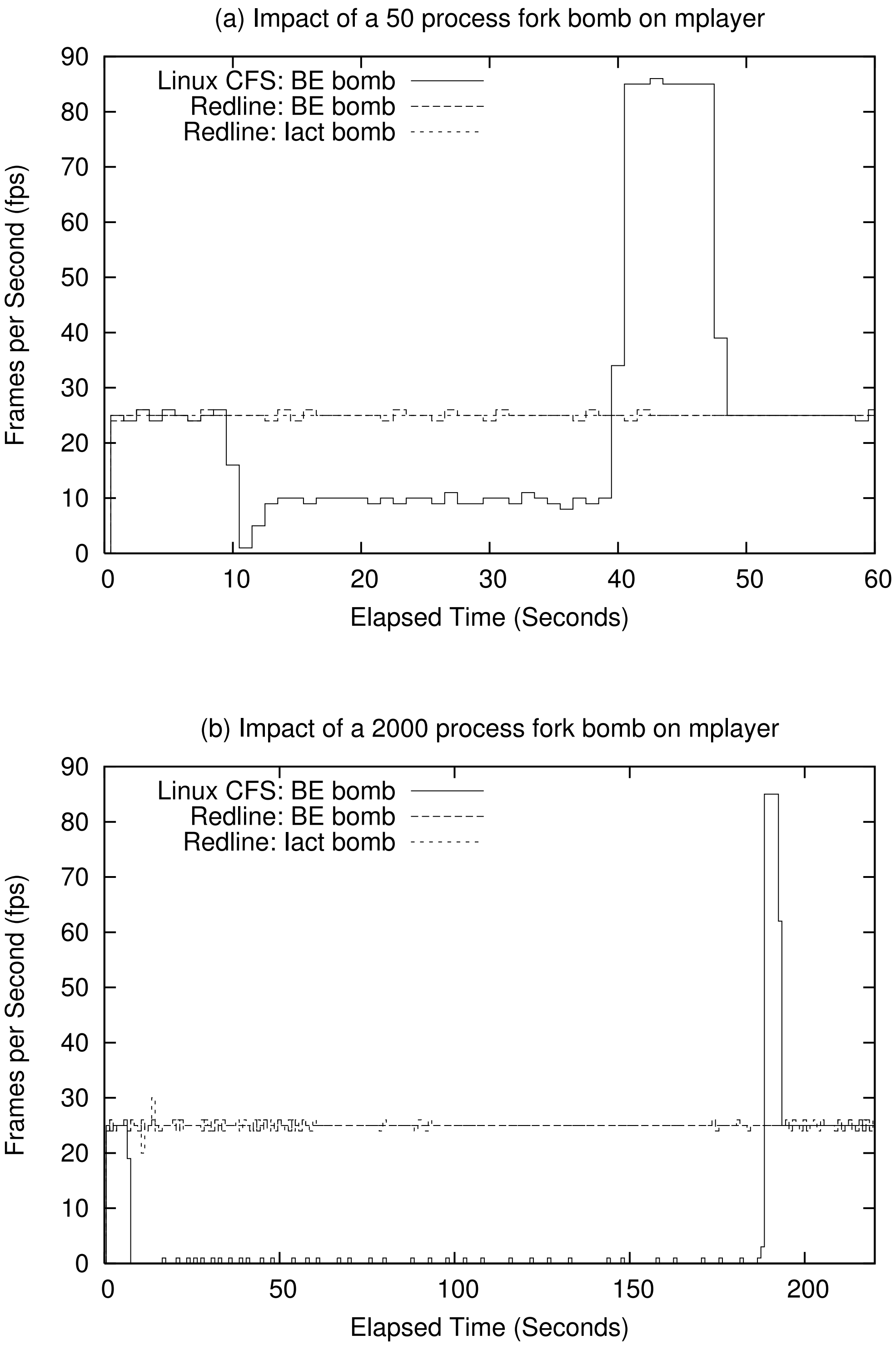


Figure 11.2. Playing a video and launching fork bombs of (a) 50 or (b) 2,000 CPU intensive tasks. They immediately disrupt the responsiveness of the video in Linux, but have almost no impact in Redline

terminates at the 40 second mark, mplayer "catches up" by playing frames at more than triple the normal rate.[1]

For Linux, the 2000-task fork bomb has a similar effect on mplayer, only much worse. Figure 11.2((b) shows that mplayer receives so little CPU bandwidth that its frame rate drops nearly to zero (one frame every 4 or 5 seconds). The load is so high that even the fork bomb's parent task is unable to kill all of the children after 30 seconds, delaying the "catch-up" phase of mplayer until approximately the 190 second mark. In fact, the whole Linux system becomes unresponsive, with simple tasks like moving the mouse and switching between windows becoming so slow that human intervention is impossible.

In Redline, these fork bombs, whether run as best-effort or interactive tasks, have a negligible impact on mplayer. This indicates that Redline successfully isolates interactive tasks from best-effort ones, and its admission control effectively prevents the system from accepting too much workload. Only the 2,000 task interactive fork bomb briefly degrades the frame rate to 20 f/s, which is a barely perceptible effect. This brief degradation is caused by the 1 ms period that Redline gives to each newly forked task before performing an admission test, leaving the system temporarily overloaded.

11.2.3 Competing Interactive Tasks

In order to see how interactive tasks may affect each other, we launch mplayer, and then we have a user drag a window in a circle for 20 seconds. Figure 11.3 shows that, under Linux, moving a window has substantial impact on mplayer. Because we use a high screen resolution and a weakly powered graphics card, Xorg requires a good deal of CPU bandwidth to update the screen. However, the CFS scheduler gives the same CPU share to all runnable tasks, allowing the window manager to submit screen update requests faster than Xorg can process them. When mplayer wakes up, it has to wait until all other runnable

[1] Catching up is one of mplayer's features that can be turned off at the user's will. Once it is turned off, the spikes after the fork-bombs will disappear.

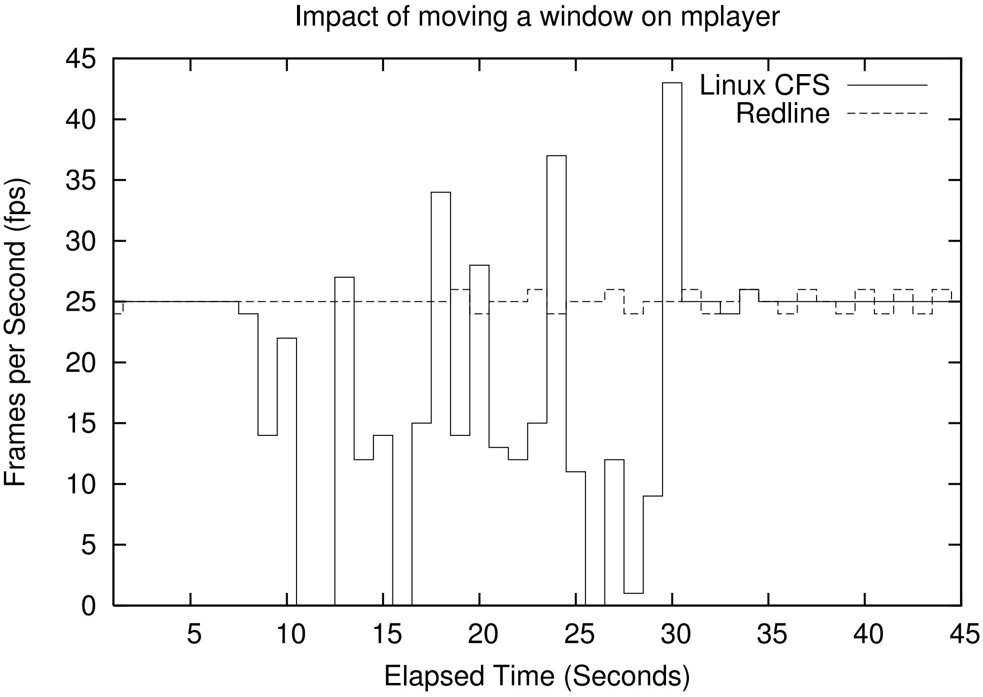


Figure 11.3. Playing video while dragging around a window.

tasks make sufficient progress before it is scheduled for execution. Moreover, its requests are inserted at the end of Xorg's backlogged service queue. Consequently, the frame rate of mplayer becomes quite erratic as it falls behind and then tries to catch up by submitting a group of frame updates in rapid succession.

In Redline, mplayer plays the movie smoothly no matter how quickly we move the window, even though Xorg and all of the tasks comprising the GUI are themselves interactive tasks. We believe that this is because Xorg effectively gets more bandwidth (50% reserved plus proportional sharing with other tasks). One by-product of Xorg's receiving more bandwidth is that the desktop or window manager receives less bandwidth, which effectively suppresses the total number of window updates submitted. Furthermore, the EDF scheduler causes mplayer add its requests into Xorg's service queue earlier, causing the frame to be played on time.

11.3 Memory Management

Redline extends the standard VM system with several mechanisms to ensure better response time for interactive applications. In this part, we evaluate how well these mechanisms work (i.e., protecting working sets, the rate-controlled reserve, and speed-bump pages) in the face of heavy memory contention.

11.3.1 Memory overload – Memory Bombs

We simulate a workload that has a high memory demand by using memory bombs. This experiment creates four tasks, each of which allocates 300 MB of heap space and then repeatedly writes to each page in an infinite loop. For Redline, we perform two experiments: the first launches the memory bombs as best-effort tasks, and the second launches them as interactive ones using specification 10:100.

The upper part of Figure 11.4 shows the frame rate for mplayer over time on Linux with the memory bomb tasks running. The frame rate is so erratic that the movie is unwatchable. Both the video and the audio pause periodically. The memory bomb forces the VMM to swap out many pages used by GUI applications, making the system as a whole unresponsive. It appears that the erratic frame rate settles somewhat at around the 80 second mark. The lower part of Figure 11.4 shows the time at which each frame is played during the period from second 118 to second 120. We see that although the overall frame rate during this period is within a normal range, frames are displayed in bursts, still leaving the user with an undesirable experience.

As shown by Figure 11.4, under Redline mplayer successfully survives both the best-effort and interactive memory bombs. Each of them leads to only one brief disruption of the frame rate (less than 3 seconds long), [2] which a user will notice but is likely to tolerate. Most of the time Redline plays the movie at a steady rate. The system remains responsive,

[2] The disruption can be dramatically reduced once logging is turned off. We modified mplayer to record the time at which a frame is played, which introduces extra memory demand and I/O requests.

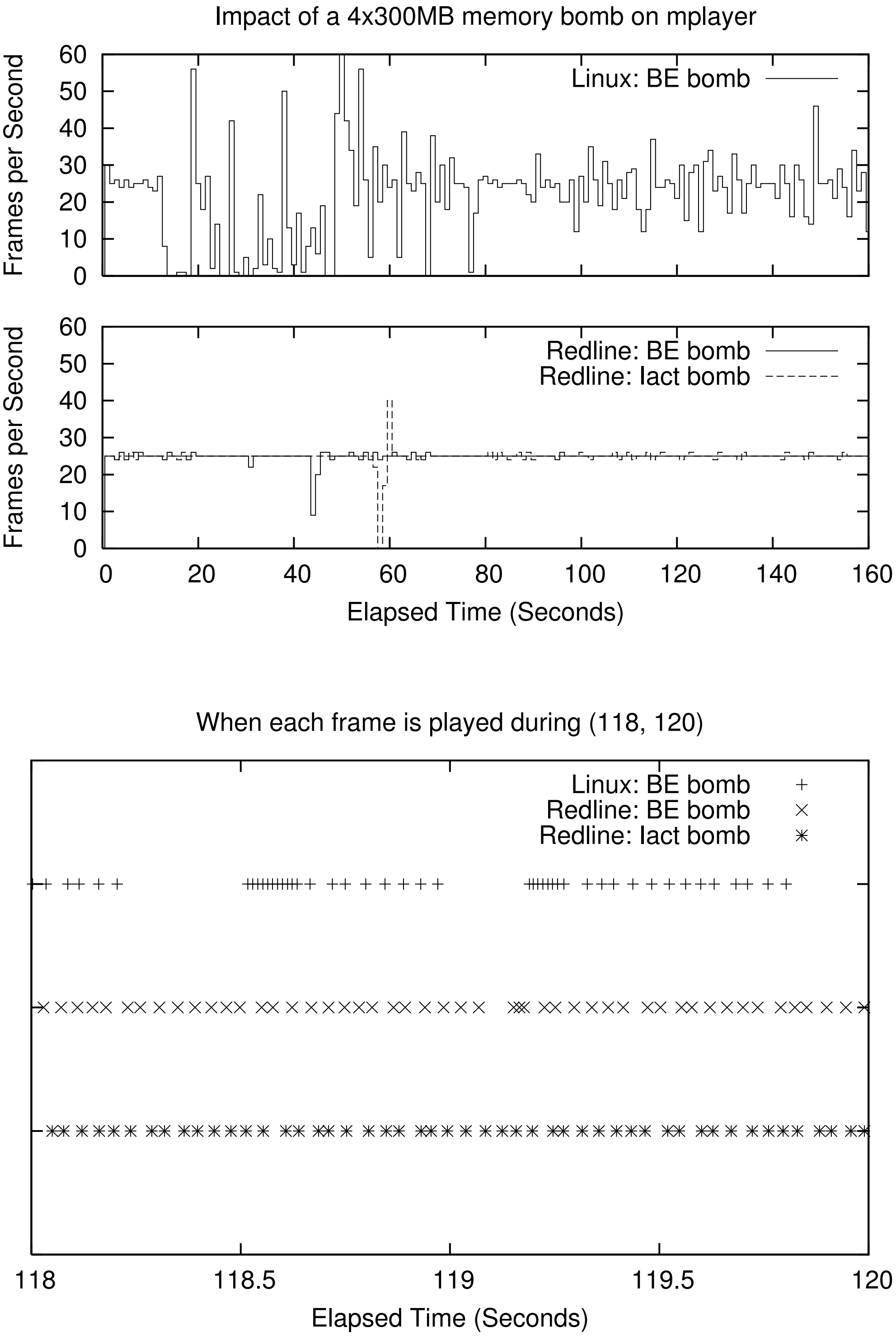


Figure 11.4. Playing video with 4 x 300 MB memory bomb tasks. The frame rate is severely erratic under Linux, but is steady under Redline. The lower part shows when each frame is played during the interval (118s, 120s).

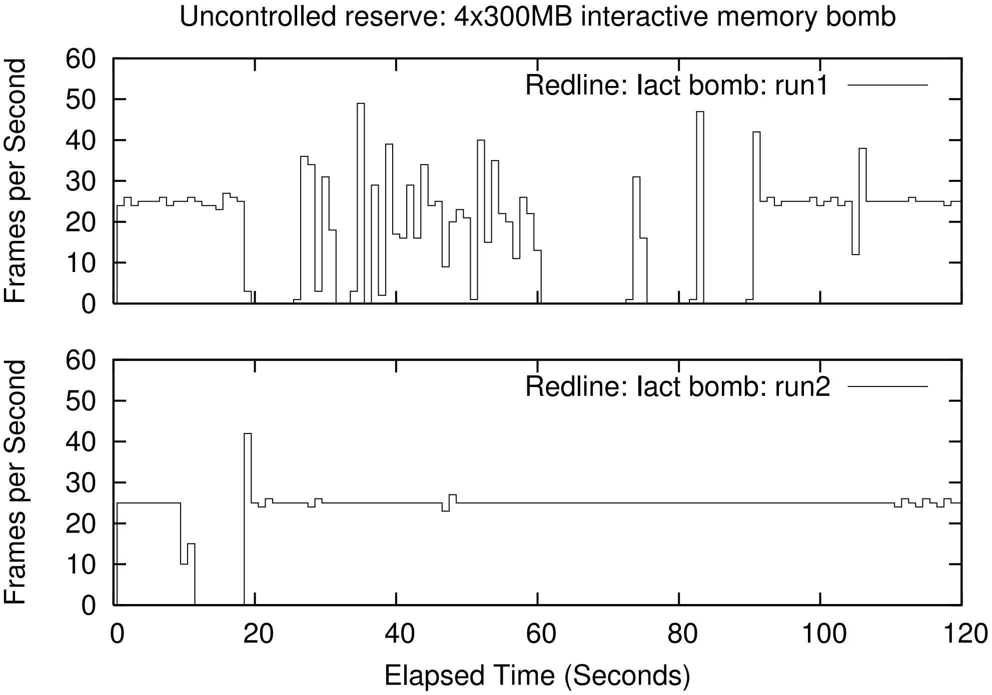


Figure 11.5. Playing a movie with 4 x 300 MB interactive memory bomb tasks. It show that, without the rate controlled memory reserve, the Redline system can be disrupted by various amount of time in the fact of memory contention.

allowing the user to carry out GUI operations and interact as usual. We additionally tried to launch firefox during the memory bomb's execution. Under Linux, this task requires four minutes to load; under Redline, it requires less than 30 seconds. While this result reveals some degradation in responsiveness under Redline, it is far less severe than under Linux.

11.3.2 Effectiveness of the Rate-controlled Reserve

In order to demonstrate the importance of the rate controlled reserve, we remove it from Redline and repeat the interactive memory bomb experiment. Figure 11.5 shows how mplayer behaves in two different runs. In the first, memory-demanding interactive tasks quickly exhaust the free memory, forcing others tasks to reclaim pages when allocating memory. Therefore, mplayer is unable to maintain its frame rate. At approximately the 90 second mark, the Redline VMM finally demotes an interactive task to the best-effort

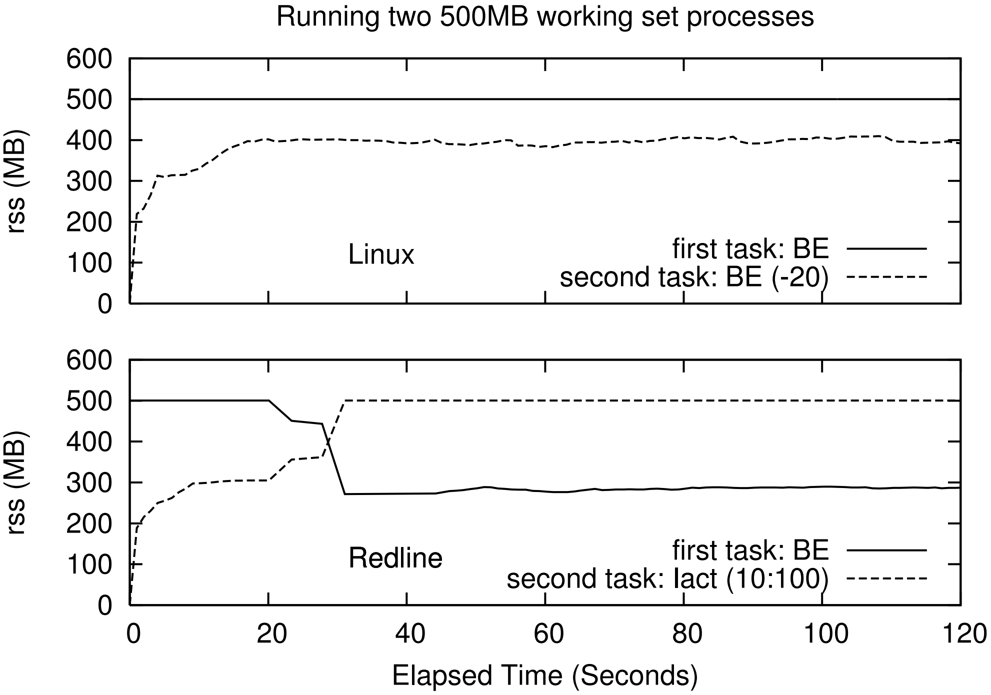


Figure 11.6. Competing memory bomb tasks. Under Linux, the lower-priority background task prevents the higher-priority foreground task from caching its working set. In Redline, the interactive task quickly builds up it working set.

class, and then the frame rate of mplayer stabilizes. Depending on when and which tasks the Redline VMM chooses to revoke, the interactive memory bomb can prevent the system from being responsive for a long period of time. Here, more than one minute passes before responsiveness is restored. However, during the second run, mplayer was unresponsive for only about 10 seconds, thanks to a different selection of tasks to demote. Ultimately, the limited isolation among interactive tasks provided by this small rate-controlled reserve is crucial to maintaining a consistently responsive system.

11.3.3 Effectiveness of Setting Speed-bump Pages

To examine the effectiveness of Redline's speed-bump page mechanism, we first start one 500 MB memory bomb task. After a few seconds, we launch a second 500 MB memory bomb. Under Linux, we set this second task's priority to be -20. Under Redline, we launch it as an interactive task whose specification is set to 10:100. Figure 11.6 presents the *resident set sizes (RSS)*—the actual number of cached pages—for each task over time. Under Linux, the second task, in spite of its high priority, is never allocated its complete working set of 500 MB. Here, the first task dirties pages too fast, preventing the Linux VMM from ever reallocating page frames to the higher priority task. However, under Redline, the second task is quickly allocated space for its full working set, stealing pages from the first, best-effort task.

As an alternative test of this mechanism, we launched four 300 MB memory bombs, and then launched the firefox web browser. Under Redline, firefox was ready and responsive after 30 seconds; under a standard Linux kernel, more than 4 minutes passed before the browser could be used.

11.4 Disk I/O Management

Finally, we examine the effectiveness of Redline's disk I/O management by verifying how interactive tasks are affected by background tasks that perform intensive disk I/O.

11.4.1 Reading

For testing disk reading operations, we play a movie using mplayer in the foreground while nine background tasks consume all of the disk bandwidth. Each background task reads 100 MB from disk in 20 MB chunks using direct I/O (bypassing the file system cache to achieve steady I/O streams, instead of filling up the page cache to trigger page swapping). Figure 11.7 shows the frame rate of mplayer over time for both Linux and Redline. Under Linux, mplayer blocks frequently because of the pending I/O requests of the background tasks, thus making the frame rate severely degraded and erratic. Redline automatically assigns higher I/O priorities to the interactive task, allowing it to maintain its frame-rate throughout.

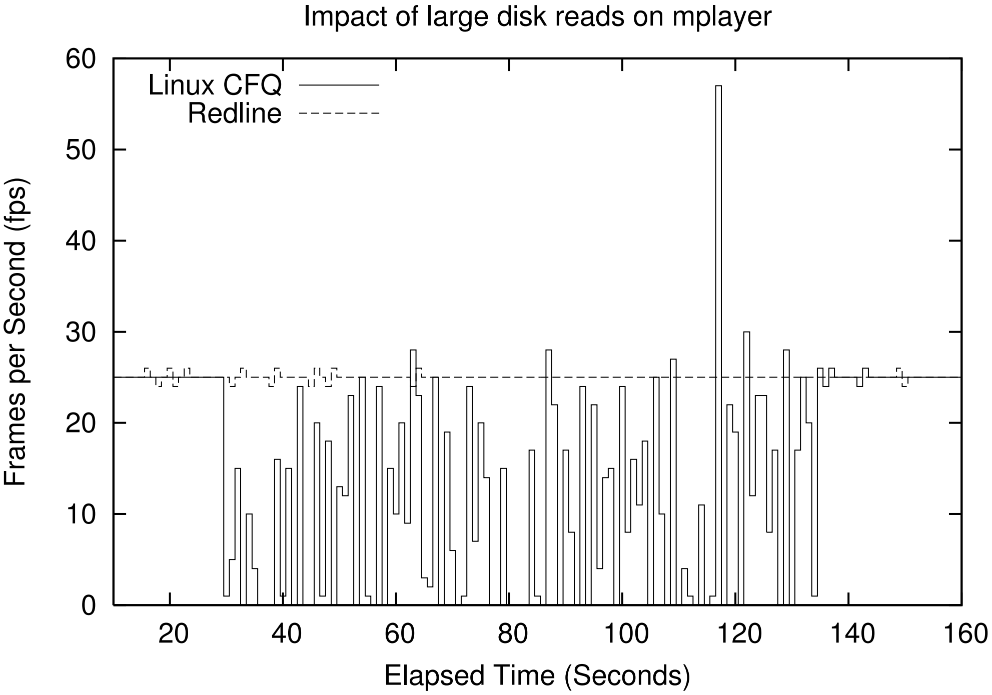


Figure 11.7. Read: massive reads cause mplayer to display the movie erratically under Linux, while playback remains smooth in Redline.

11.4.2 Writing

To test disk writing operations, we launch two background tasks meant to interfere with responsiveness. Specifically, each task repeatedly writes to an existing 200 MB file using three different modes: 1) buffered writes (BW); 2) large direct I/O writes with a 200MB buffer (DW200MB); 3) small direct I/O writes with a 1 page buffer (DW1Page).

We then use vim, modified to report the time required to execute its write command, to perform sporadic write requests of a 30 KB file. We also use a program iowrite to save a 100 MB file. Both of them are set to the highest priority (-20) under Linux, while they are launched as interactive tasks with a specification of 5:100 under Redline. Figure 11.8 present the results of these experiments, with vim on the left side and iowrite on the right side.

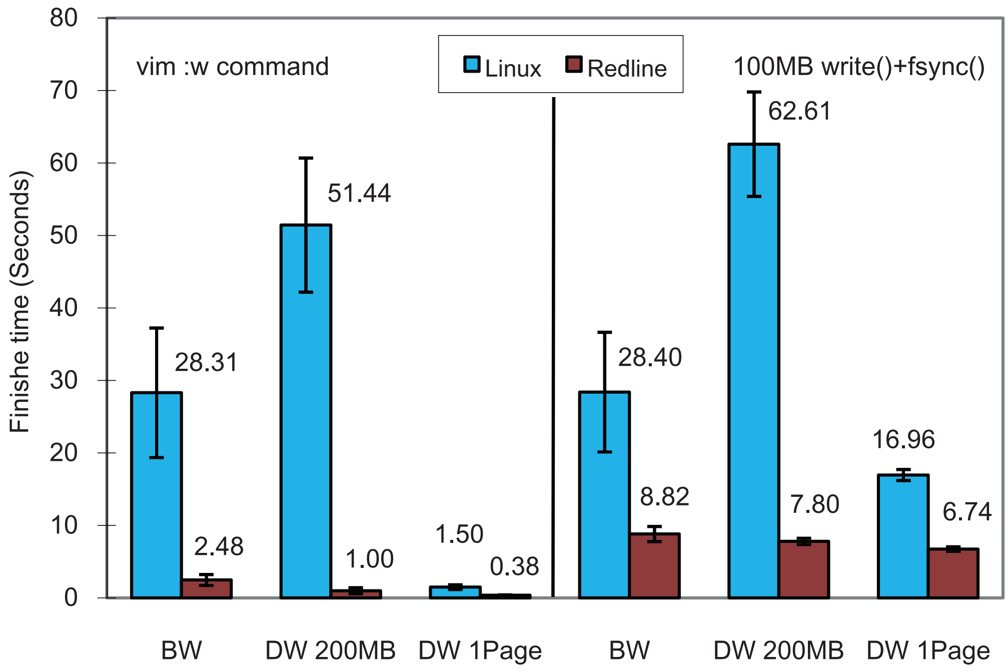


Figure 11.8. Write: saving 30 KB in vim or writing 100 MB with a background workload writing to disk in three modes. Bad interactions among different components can degrade responsiveness dramatically, and Redline successfully eliminates their effects.

Let's first examine buffered writes (BW). For Linux, each transaction in the journaling file system is heavily loaded with dirty pages from the background tasks. Thus, the calls to fsync() performed by vim cause it to block for a mean of 28 seconds, which is almost the same as saving a 100MB file. Under Redline, the reduced dirty threshold for best-effort tasks forces the system to flush the dirtied pages of the background task more frequently. When vim calls fsync(), the transaction committed by the journaling file system takes much less time because it is much smaller, requiring a mean of only 2.5 seconds.

For the DW200MB case, Linux performs even worse than buffered writes, because huge direct I/O operations can freeze the current transaction, blocking later I/O requests for long periods of time. On average, vim need 61 seconds to save the file. In Redline, large direct I/O operations are broken into smaller pieces, reducing the amount of time needed to about 1.0 seconds. For the small direct I/O (DW1Page) case, Redline still beats

Linux, because it automatically gives higher priority to I/O requests from interactive tasks. The results for iowrite are qualitatively similar to those for vim.

CHAPTER 12

REDLINE FOR MULTI-THREADED SERVERS

We have presented how Redline can be used to maintain system-wide responsiveness in desktop environments. In this part of the dissertation, we further explore how multi-threaded sever applications can make use of the Redline system to achieve better response time guarantees. As usual, we start by describing the inadequacy of existing operating systems, then propose the Redline-based server architecture, and finally present a proof-of-concept evaluation.

12.1 Introduction

Most server applications are now multi-threaded. Multi-threading allows a server to process multiple requests simultaneously, achieving higher throughput by making better use of resources (e.g., exploiting multiple CPUs and hiding I/O latencies). Many server applications are also response-time sensitive, such as game servers and web servers. However, the proportional-share schedulers used by existing operating systems fail to allow these multi-threaded server applications to choose an appropriate number of threads, especially for handling a large burst of requests. Too many threads reduces the CPU bandwidth received by each of them, which leads to long response time. Too few threads can fail to hide I/O latencies, causing low throughput.

Proportional-share schedulers attempt to ensure that every thread makes equal progress by evenly distributing CPU bandwidth to them, which is oblivious to response time requirements. If a server uses too many threads, it will have a very long response time in the face of a large burst of incoming requests.

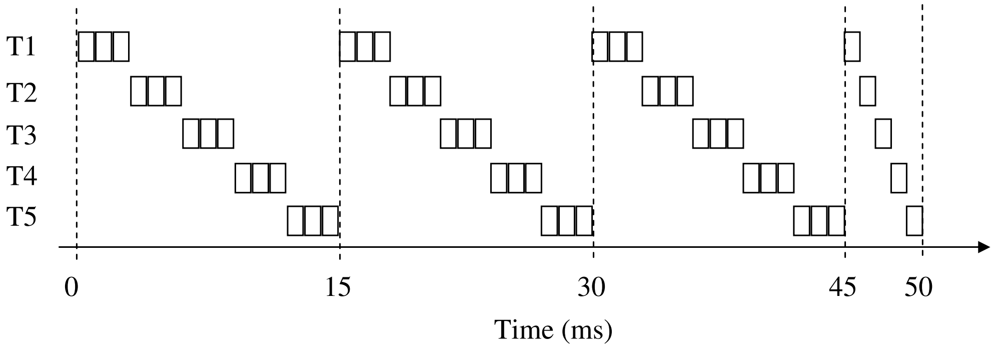


Figure 12.1. How the Linux CPU scheduler uses 5 threads to serve 5 requests, where each request performs 10ms computation work. All requests finish at the end of the 50ms period due to proportional sharing.

We demonstrate this problem with Linux's CFS scheduler, whose default scheduling quantum is 3 ms. Consider the case where a program uses 5 threads to serve 5 requests, each with 10 ms computation work. As shown in Figure 12.1, the CFS scheduler executes each thread for 3 ms in turn, and all requests finish at the end of the 50 ms period. Similarly, when 200 requests are processed by 200 threads simultaneously using the CFS scheduler, all of them will finish at the end of the 2 second period. If the response time requirement for these requests is 1 second, then they all miss the deadline, even though 100 of them could be satisfied. Clearly, the more threads we use, the less CPU bandwidth each of them receives, and thus the longer response time for each request.

Even worse, a proportional-share scheduler can sometimes also degrade throughput when disk I/O is involved. We modify the previous example: after 10 ms of computation work, each request issues a disk I/O that needs about 5 ms to complete. Figure 12.2 demonstrates what would happen under Linux. At the time when the first disk I/O is issued, the system has already performed most of the computation work of these requests, leaving very little to cover the disk I/O latencies. In other words, it fails to overlap disk I/O latencies with useful computation work, resulting in lower throughput. In this example, with careful

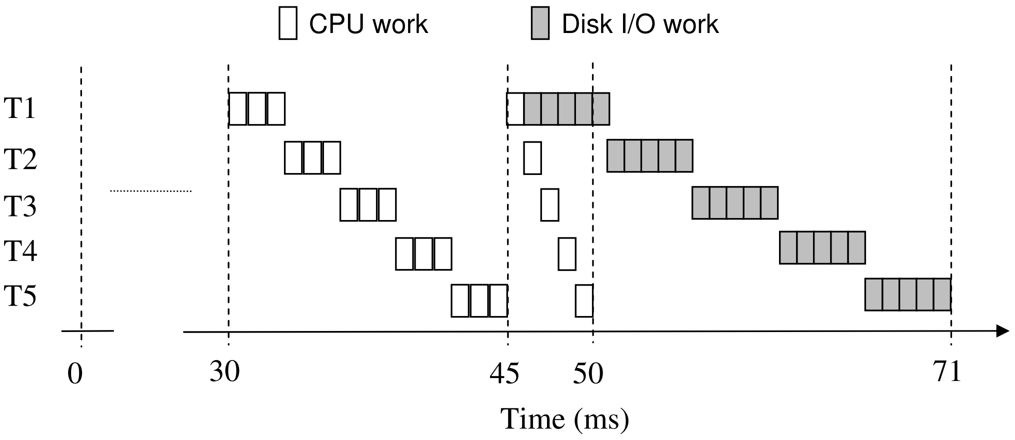


Figure 12.2. How Linux CPU scheduler uses 5 threads to serve 5 requests, where each request performs 10ms computation work and then issues a disk I/O. Due to proportional-sharing, when disk I/O starts, there is no enough computation work left to hide their latencies, resulting in lower throughput.

arrangement, the system should be able to process these requests in 55 ms, rather than 71 ms.

In general, a server application should choose a number of threads that is large enough to maximize the chances of hiding I/O latencies, but small enough to avoid compromising response time guarantees. This is quite challenging since requests may have varying workloads. It is difficult for a server application itself to make the right choice dynamically without appropriate support from operating system. However, as we have already described, existing operating systems do not provide this kind of support, because their CPU schedulers are non-cooperative and oblivious to response time requirements.

Redline's specification-driven scheduler has the ability to handle requests with response time requirements and to isolate them from best-effort ones. A server application can dynamically calculate the number of requests it can simultaneously support, based on the information provided by the underlying CPU scheduler (e.g., average CPU time). It then chooses the specifications for these requests according to the given response time requirement, and feeds them to the CPU scheduler to control request processing. Through this

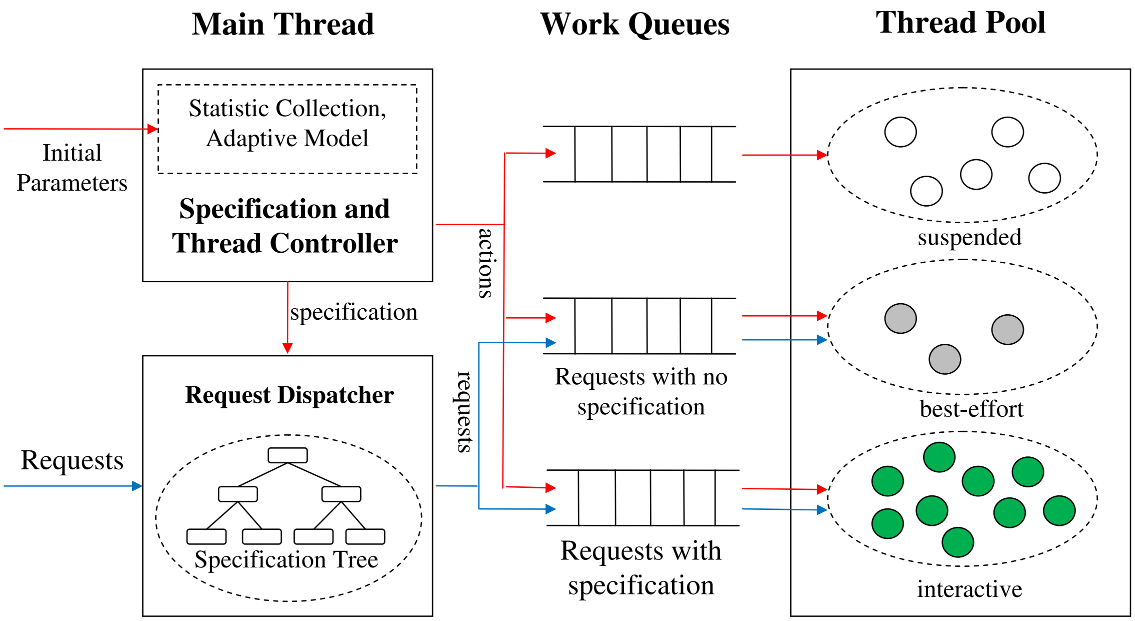


Figure 12.3. Redline server architecture.

cooperation, a server application can maximize the number of requests that meet the desired response time requirement while maintaining high throughput, even when incoming requests overload the system.

12.2 Redline Server Architecture

Figure 12.3 presents an overview of the architecture of a Redline server that cooperates with the underlying Redline system. The main thread of a Redline server has two important components: the *specification and thread controller* and the *request dispatcher*. The controller uses the workload information from the underlying operating system and user-supplied response time requirements to determine the number of requests the server can simultaneously support (i.e., the number of interactive threads needed), as well as their specifications. The dispatcher, upon receiving a request, searches through the specifications. It attaches the first available specification to the request, and activates an interactive thread to process the request. If it does not find a specification, it simply hands over the request to a best-effort thread. Before processing a request, an interactive thread first informs the operating system with the specification attached, so that it can be properly executed by Redline's EDF scheduler.

To make later discussion easier, we first present a list of parameters used by the Redline server, and what they stand for:

- P : the percentage of CPU bandwidth dedicated to interactive threads,
- RT : the desired response time for processing a request, in milliseconds,
- C_{avg} : the average CPU time for processing a request, in milliseconds
- C_{max} : the CPU time large enough for processing 95% of requests,
- N : the number of requests that can be served in every period of RT,
- N_{BE} : the number of best-effort threads, and
- N_{Iact} : the number of interactive threads.

12.2.1 Manging Specifications and Threads

The Redline server takes advantage of the EDF scheduler in the underlying Redline system to serve requests with response time requirements. It is crucial that it not accept too many requests, and thus overload the CPU bandwidth allocated to interactive threads. Therefore, it keeps track of the average CPU time of processing a request (C_{avg}), and finds the largest N that satisfies the following equation:

$$N \times C_{avg} < RT \times P \tag{12.1}$$

Intuitively, this equation says: within any period of RT, the total amount of computation work from accepted requests should not be more than the CPU bandwidth allocated to interactive threads. The Redline server allocates exactly N specification objects to match the number of requests it can serve simultaneously. For each of them, it sets their CPU reservation $C : T$ to be:

- Reserved amount (C): $C \leftarrow max(C_{avg}, C_{max})$
- Response period (T): $T \leftarrow RT$

To ensure that the server does not accept an excessive amount of work, a specification object should not be used more than once during any period of RT. Otherwise, Equation 12.1 would no longer hold. Therefore, each specification also has a *startTime*, and the dispatcher considers it *eligible* for use only if the current time $now \geq startTime$. Once a specification is used, its next *startTime* is set to be $now + RT$. In our implementation, specifications are organized as a red-black tree, sorted by *startTime*.

When a request arrives, if there is an eligible specification, the dispatcher will accept the request and hand over the request (together with its specification) to an interactive thread. Otherwise, the request is given to a best-effort thread. Hence, the Redline server maintains a pool of N interactive threads (i.e., $N_{Iact} = N$), and each processes exactly one request once waked up.[1] In this sense, the Redline server is request oriented, because it applies specifications to requests, not directly to threads. In fact, what is important here is how the dispatcher gives out specifications, rather than the number of interactive threads used.

Due to the admission and load control in the underlying Redline system, the Redline server cannot dedicate more than R_{hi} (i.e., 90%) CPU bandwidth to interactive threads. Therefore, it also maintains a small pool of best-effort threads (e.g., $N_{BE} = 10$), so that the server can make use of any remaining CPU bandwidth, and process requests that are not accepted.

[1] Note that the number of interactive threads needed is not always exactly N. When the request arrival rate is low and the system is underloaded, these threads can be reused. The dispatcher only activates a new interactive thread when all interactive threads are busy. Thus, if the server is fully loaded, it needs N interactive threads.

12.2.2 Serving a Request

Upon receiving a request, the dispatcher performs a search on the specification tree, looking for the first eligible specification. If none are eligible, the request is simply appended to the end of the best-effort workqueue. Otherwise, the dispatcher removes the specification from the tree, sets its next *startTime* time to be $now + RT$, and binds it with the request. Finally, the dispatcher inserts the request into the interactive workqueue and wakes up an interactive thread.

After grabbing a request from the workqueue, an interactive thread first retrieves the corresponding specification and informs the underlying EDF scheduler to get a new start time and a period. It then starts to process the request. When done, it collects necessary information (e.g., the amount of CPU time consumed by processing the request), inserts the specification back in the tree, and finally puts itself back to sleep, waiting for the next request.

12.2.3 Adapting to Changes

The Redline server keeps a history of the amount of CPU time consumed for processing each request, so that it can dynamically adapt to workload changes or response time requirement changes. Any of the following events can trigger an adjustment:

- The user changes the response time requirements (RT).
- The user changes the CPU bandwidth dedicated to interactive threads (P).
- A certain number of requests has been processed since the last adjustment.
- A certain amount of time has passed since the last adjustment.

For each adjustment, the controller first re-calculates C_{avg} and C_{max} based on the history information preserved since the last adjustment. It then determines the new N using Equation 12.1 and configures the server to have a matching number of specifications and threads. In order to do so, the controller adds an appropriate number of *action* requests into the workqueues. Idle threads are then awakened to change their type, and busy threads are not affected. Finally, it updates the CPU reservation used by these specifications, if necessary.

12.3 Proof-of-concept Evaluation

We have implemented a prototype of the Redline server that works cooperatively with our Redline system. We evaluate our implementation on the same hardware platform that we used in evaluating the Redline system, and compare it against a simple server running on Linux that is implemented using a thread pool.

Experiment settings: We feed the server (both Redline and Linux version) with a burst of 200 requests, and set the response time target to be 1 second (1000 ms). We consider two cases: a *CPU intensive* case in which each request triggers the server to perform about 10 ms computation work, and a *mixture of CPU and I/O* case in which each request has a disk I/O after the computation is finished. For performing the disk I/O, the server side keeps a pool of 500 one-page files, and every request randomly asks the server to load the content from one of them using direct I/O. Clearly, this setting overloads the system, and thus not all requests can be satisfied with respect to the 1 second response time requirement.

The Linux server is configured to process the request burst using 1, 10, 50, 100, and 200 threads. The Redline server is configured to use 90% of the CPU bandwidth for interactive threads (the default R_{hi} for Redline) and chooses the number of threads automatically. As a result, it uses 86 interactive threads and 10 best-effort threads, and 11:1000 as the specification for them. Table 12.1 summarizes the results of our experiments. For each of them, we present the number of requests that are satisfied and the total amount of time elapsed before all the requests are processed.

For the CPU intensive case, the Linux server with 1 thread performs the best, satisfying 98 requests out of 200. Our Redline server satisfies 96, which is nearly optimal. Both of

	CPU			CPU + I/O		
	Threads	Req. Satisfied	Total Time (ms)	Threads	Req. Satisfied	Total Time (ms)
Linux Server	1	98	2035.3	1	69	2677.0
	10	91	2037.5	10	86	2112.8
	50	67	2041.0	50	54	2064.1
	100	37	2041.3	100	2	2522.3
	200	0	2039.1	200	0	3347.9
Redline Server	86/10	96	2038.9	85/10	91	2053.3

Table 12.1. Using the Linux server and the Redline server to process a burst of 200 requests, each with about 10 ms computation work, and whose response time requirement is 1 second. We also consider a case where the computation is followed by a random disk I/O. In both cases, the Redline server is able to satisfy more requests while maintaining high throughput.

them have similar throughput, given that using more threads only adds slight overhead. However, once disk I/O is involved, the Linux server performs much worse. With 10 threads, it satisfies more requests but has a lower throughput. With 50 threads, it gets the best throughput, but satisfies far fewer requests. Our Redline sever offers the best combination of responsiveness (91 requests) and throughput (2053 ms). Adding an extra I/O operation does not have any significant impact on server performance. To summarize, the Redline server is able to maximize the number of requests satisfied while maintaining high throughput and supporting more threads.

12.3.1 CPU Intensive

We next take a closer look at how the Redline and Linux servers process these requests. Figure 12.4 shows exactly when each request finishes for the CPU intensive case.

The Linux server with 1 thread processes the requests one by one, spreading their finish times evenly over the entire period. Therefore, the number of requests it satisfies is optimal. As the number of threads increases, each thread receives less CPU bandwidth due to proportional-share scheduling, and the number of requests satisfied starts to drop. In fact, requests tend to finish in clusters, whose size is approximately the number of threads used.

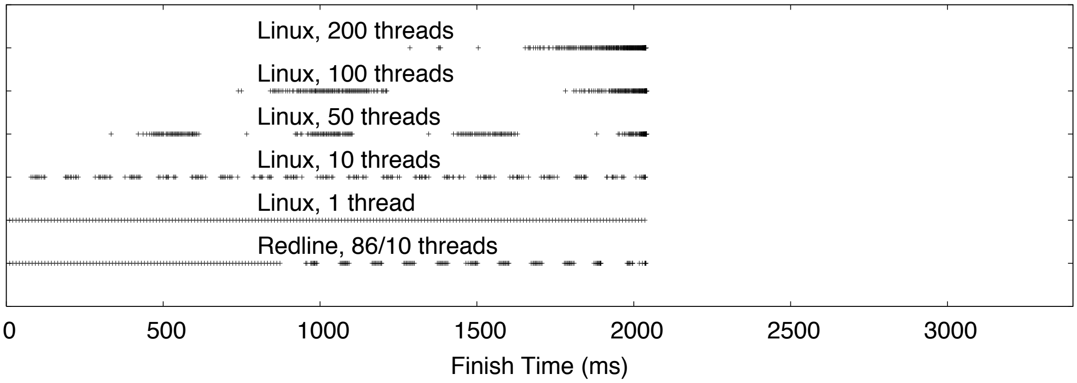


Figure 12.4. Comparing the Redline and Linux servers (CPU only). The Redline server achieve nearly optimal responsiveness (as does the Linux server with 1 thread) without sacrificing throughput. Proportional-share scheduling can cause all requests to miss their deadline, as happens when the Linux server uses a large number of threads.

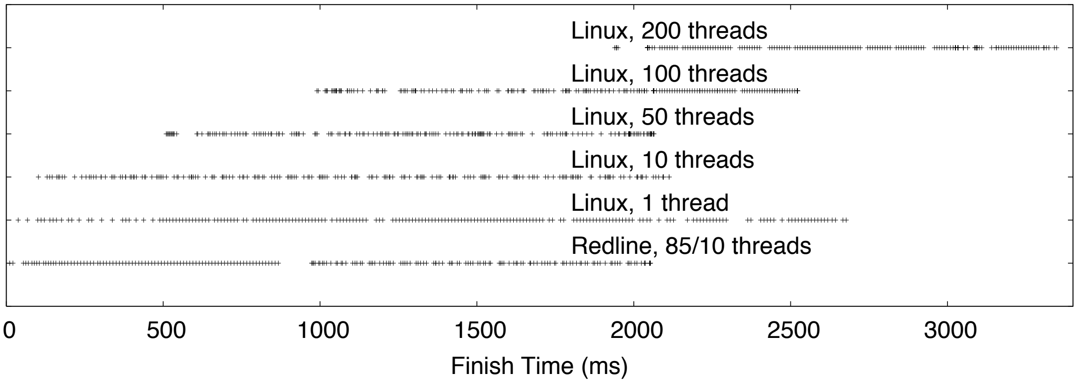


Figure 12.5. Comparing the Redline and Linux servers (CPU + I/O). An additional disk I/O has almost no impact on the Redline server. It offers the best combination of responsiveness and throughput (almost the same as in the CPU-only case) in all the configurations tested.

The Linux server with 200 threads performs the worst, since all requests finish towards the end and miss their 1 second deadline.

The Redline server automatically calculates the number of requests it can satisfy based on the actual workload, and uses interactive threads to process them. The remaining requests are served by a small pool of best-effort threads (10 in our experiments). As a result, it behaves very much like the combination of the Linux server with 1 thread and with 10 threads. During the first half, all interactive threads are lined up according to their deadlines. Redline's EDF scheduler executes these threads one by one, where each processes one request. After that, a small set of best-effort threads, handled by the proportional-share scheduler, processes the remaining requests.

12.3.2 Mixture of CPU and I/O

Similarly, Figure 12.5 shows exactly when each request is finished if they perform an additional disk I/O after computation. It shows that the extra disk I/O has a huge impact on the Linux server's performance, but not on our Redline server.

Interestingly, the Linux server with 1 thread performs badly, but is not the worst. We believe that this is because the computation work between I/O operations statistically makes the disk head closer to the location of the next random disk access, and thus hides part of its rotational latency. Using a small set of threads increases performance substantially, since it hides I/O latencies better. However, as the number of threads further increases, the effects of proportional-share scheduling start to dominate, and both the responsiveness and throughput drop quickly. The worst case happens when serving 200 requests with 200 threads. This is because, at the time when disk I/O starts, there is almost no computation work left, and thus the mechanical latencies are completely exposed.

The result for our Redline server is quantitatively and qualitatively similar to that of the CPU intensive case. During the first half, the Redline server uses interactive threads to process requests. Again, the number of such requests is determined based on the actual workload and the response time requirement. The Redline's EDF scheduler executes the next thread only when the previous one is blocked on I/O, and thus fully covers its I/O latency with computation work from later requests. Once all interactive threads finish, a small set of best-effort threads process the remaining requests, which has been proven to be a good choice under proportional-share scheduling. This combination allows the Redline server to satisfy more requests than any Linux server, while preserving high throughput.

CHAPTER 13

RELATED WORK

13.1 Overview

In this chapter, we investigate the research has been done on managing CPU, memory, and disk I/O, focusing on the areas that are most relevant to our work, and discuss how our proposed approach may contribute these areas.

We first present related work on automatic heap size adjustment for garbage collected applications (Section 13.2). We then discuss CPU scheduling, including general purpose systems (Section 13.3), real time systems (Section 13.4), and CPU scheduling frameworks (Section 13.5). Section 13.6 presents related work on memory management, in particular working set tracking and page replacement mechanisms. Section 13.7 covers disk I/O management. Finally, Section 13.8 discusses prior integrated resource management approaches.

13.2 Automatic Heap Sizing for Garbage-collected Applications

Researchers have proposed a number of heap sizing approaches for garbage collection; Table 13.1 provides a summary. The closest work to CRAMM is by Alonso and Appel, who also exploit VM system information to adjust the heap size [5]. Their collector periodically queries the VM for the current amount of available memory and adjusts the heap size in response. CRAMM differs from this work in several key respects. While their approach shrinks the heap when memory pressure is high, it does not expand and thus reduce GC frequency when pressure is low. It also relies on standard interfaces to the VM system that

	Grow Heap	Shrink Heap	Stat. Alloc	Dyna. Alloc	GC Neut	OS Supp	Responds to
Alonso et al.[5]		√	√	√		√	memory allocation
Brecht et al.[28]	√		√				pre-defined rules
Cooper et al.[43]	√		√				user supplied target
Page-Level [136]	√	√	√	√			page swapping
Isla Vista [62]	√	√	√	√		√	stalls during allocation
BC [67]		√	√	√		√	page swapping
JRockit [19]	√	√	√		√		throughput/pause time
HotSpot [69]	√	√	√				throughput/pause time
MMTk [23]	√	√	√		√		live ratio and GC load
CRAMM/AHS [135]	√	√	√	√	√	√	**memory allocation**

Table 13.1. A comparison of approaches to dynamic heap sizing.

provide a coarse and often inaccurate estimate of memory pressure. The CRAMM VM captures detailed reference information and provides reliable values.

Brecht et al. adapt Alonso and Appel's approach to control heap growth via ad hoc rules for two given *static* memory sizes [28]. Cooper et al. dynamically adjust the heap size of an Appel-style collector according to a *user-supplied* memory usage target [43]. If the target matches the amount of free memory, their approach adjusts the heap to make full use of it. However, none of these approaches can adjust to dynamic memory allocations. CRAMM automatically identifies an optimal heap size using data from the VM. Furthermore, the CRAMM model captures the relationship between working set size and heap size, making its approach more general and robust. Zhang et al. [136] adjust the heap size in small increments in response to the number of page faults encountered at application phase boundaries, making the adaptation slow. Similarly, Isla Vista [62] uses *allocation stalls* reported by the VM as feedback to trigger garbage collection and adjusts the heap size gradually, making the adaptation slow also.

The bookmarking collector (BC) is a garbage-collection algorithm that guides a lightly modified VM system to evict pages that do not contain live objects and installs "bookmarks" in pages in response to eviction notifications [67]. These bookmarks allow BC to collect the heap without touching already evicted pages, which CRAMM must. One shortcoming of BC is that it currently cannot grow the heap because it responds only to page eviction notifications. CRAMM both shrinks and grows the heap to fit, and can be applied to a wide range of existing garbage collection algorithms.

13.3 CPU Scheduling for General Purpose Systems

Most existing general purpose operating systems put most of their focus on scheduling best-effort applications. Their schedulers fall mainly into two categories: *time-sharing* schedulers (sometimes also referred to as *fair share* schedulers) and *proportional share* schedulers. However, the distinction between these two kinds of schedulers is not clear, because ultimately they both attempt to provide proportional sharing of the CPU among applications. Furthermore, these schedulers do not address response time requirements and must be extended to support interactive applications better.

13.3.1 Time-sharing Schedulers

Priority-based fair-share schedulers arose from the need to provide proportional sharing among users in a way compatible with UNIX-style time-sharing systems [66, 77, 56]. They are still used by many general purpose operating systems, including Linux [27], FreeBSD [91], Solaris [90] and Windows [112]. Most of these schedulers use multi-level feedback with a set of priority queues. Each task has a priority that is adjusted as it executes. The feedback mechanisms used by these schedulers vary. Linux, FreeBSD, and Windows adjust priorities based on a task's execution time and average sleep time, while Solaris uses a predefined priority table. Priority-based schedulers offer different shares with respect to a task's priority by effectively running tasks at different frequencies. Usually, a priority boost is given to I/O bound tasks to provide better responsiveness.

Weighted round robin (WRR) is widely used in both network packet scheduling (e.g., DRR [114], SRR [63], StRR [108]) and CPU scheduling (e.g., charge-based [88]). Each task has a weight indicating its desired CPU share. The scheduler assigns each task a time slice whose length is proportional to its weight. As opposed to priority-based schedulers, a WRR scheduler achieves proportional share by running tasks with a different amount of time in each turn.

The lottery scheduler [130] proposed by Waldspurger associates each task with a share. Each task holds a number of lottery tickets in proportion to its share. At the beginning of each time slice, a lottery is held and the task holding the winning ticket is selected for execution. In this way, tasks running in the system statistically receive CPU cycles proportional to their shares in the long run.

Heuristics are often applied to time-sharing schedulers to handle interactive tasks better, which makes the approaches ad hoc and may lead to a phenomenon called the *interactivity trap* [128]. Human-Centered Scheduling (HuC) [57] identifies interactive tasks by monitoring I/O devices and boosts their priorities, because interactive applications usually have frequent pauses for user input. Windows Vista introduces a new scheduling class, MM_CLASS. It boosts the priorities of tasks registered in this scheduling class into the real time region (16–30) for better performance guarantees. The scheduler also monitors the CPU usage of tasks in this class. If a task uses too much, its priority is lowered back into the normal range (0–15).

These time-sharing schedulers provide only a reasonable proportional share over relative large periods. Their proportional fairness guarantees are difficult to formulate, especially in the short term. In order to provide applications with a desired performance guarantee, the user has to assign a proper priority to each application. This is extremely difficult, especially in an open environment where applications may dynamically enter and leave the system at any time. Overall, time-sharing schedulers are suitable for sharing CPU bandwidth among batch applications, but do not provide adequate support for handling response-time sensitive applications.

13.3.2 Proportional Share Schedulers

In order to overcome the drawbacks of time-sharing schedulers, researchers have developed scheduling algorithms that can provide precise proportional shares, even within a short period of time.

13.3.2.1 Fair Queuing Schedulers

Parakh and Gallagher proposed the generalized processor sharing model (GPS) [105] for analyzing network packet scheduling algorithms. In this ideal fluid model, each task has a weight. All runnable tasks are granted bandwidth proportional to their weights and execute simultaneously (i.e., scheduling happens conceptually at infinitely small intervals). The GPS model forms the basis of later developed proportional share schedulers, both for network packet scheduling and CPU scheduling.

In a real system, the resource is allocated in discrete time quanta and it is not possible for a task always to receive exactly the service time to which it is entitled (as in the GPS model). Therefore, the difference between the service time that a task should receive at a time t, and the service time it *actually* receives in a scheduling algorithm is called its *lag*, which sometimes is also referred to as *allocation error*. A negative lag means the task receives more than it should and a positive lag means it receives less than it should. The main focus of developing a proportional share scheduler is providing tight bounds on lag. The smaller the lag, the better, since it captures both allocation accuracy as well as fairness.

Demers et al. were the first to apply this notion of fairness to network packet scheduling [45]. The algorithm they proposed is called Weighted Fair Queueing (WFQ), which is also referred as Packet-to-packet Fair Queueing (PFQ). Later, Waldspurger et al. extended the same notion to CPU scheduling in their stride scheduler [131]. Since then, a number of fair queueing schedulers have been developed in the context of both network packet scheduling (e.g., SCFQ [59], Time-shift FQ [42], and BSFQ [36]), and CPU scheduling (e.g., SFQ [61], SFS [34], and EEVDF [123, 124]). Most fair queueing schedulers are based on the concept of *virtual time* introduced by Zhang [137], which serves as a measure of the degree to which a task has received its proportional share relative to other tasks. The scheduler keeps a virtual time for each task. When a task executes, its virtual time advances at a rate inversely proportional to the task's weight. The scheduler also maintains a system-wide virtual time which is advanced inversely proportional to the *total* weight of all runnable tasks. With the virtual time in hand, a fair queueing algorithm can be either deadline driven or start-time driven.

Deadline-Driven Fair Queueing: WFQ is a typical example of a deadline-driven fair queueing scheduler. It schedules tasks using a uniform time unit, the *quantum*. Given a task's current virtual time, the task's *virtual finish time* (VFT) is defined as the virtual time the task would have after executing for one quantum. WFQ maintains an ordered queue of runnable tasks sorted by VFT in increasing order, and schedules tasks by selecting the task with the smallest VFT. WFQ provides an $O(1)$ bound for the positive lag, but the bound for negative lag can be as large as $O(n)$, where n is the number of runnable tasks.

Worst-case Fair Weighted Fair Queueing (WF^{2}Q) proposed by Bennett and Zhang [21, 80] extends WFQ with eligibility control, so that the scheduler considers a task not eligible for execution if its virtual time is larger than the current system virtual time. Intuitively speaking, a task that receives more service time than it should in the GPS model is not eligible for getting more service time. The eligibility control effectively breaks the work of higher priority tasks into smaller pieces and interleaves them with the work of lower priority tasks. Therefore, WF^{2}Q offers a tight $O(1)$ lag, which is the size of a quantum.

Almost in parallel, Stoica et al. developed EEVDF [123, 124] in the context of CPU scheduling, which also provides $O(1)$ lag by using the same eligibility control. However, EEVDF goes one step further to support non-uniform quanta. Instead of simply using the uniform quantum selected by the scheduler, EEVDF allows each task to specify a CPU request size and uses it to calculate the virtual deadline of a task.

Start-Time Driven Fair Queueing: SFQ [61] sorts runnable tasks by their virtual start time and always selects the task with the smallest start time for execution. Its lag is $O(n)$ in comparison to an ideal GPS system. Chandra et al. showed that SFQ can become not work conserving in a multiprocessor system, and proposed the Surplus Fair Scheduler [34] to solve the problem by adjusting weights across different processors. From kernel version 2.6.23.1, Linux switched to a start-time driven fair queueing scheduler, called the Completely Fair Scheduler (CFS) [97]. It can be considered to be a simpler version of SFQ that does not support hierarchical scheduling. Therefore it has the same $O(n)$ lag.

13.3.2.2 Proportionate Fair Schedulers

Proportionate fair schedulers form another class of proportional share schedulers. P-fairness is based on the notion of proportionate progress proposed by Baruah et al. [15], which is essentially a discrete variant of GPS intended for running periodic tasks on multiple processors. In this model, each task in the system is associated with a share, and the scheduler tracks the progress of each task using a method very similar to that of GPS-based schedulers. An scheduling algorithm is *P-fair* if, at any instant, no task is more than on quantum away from its due share (i.e., has a tight $O(1)$ lag of one quantum).

The first proportionate fair scheduling algorithm is called *pseudo-deadlines* (PD) [17]. PD achieves P-fairness only in an ideal multiprocessor system: 1) the scheduling points of all processors are synchronized; 2) all processors use a uniform scheduling quantum; and 3) the set of tasks running in the system is fixed (i.e., tasks do not dynamically join/leave the system). PD explicitly requires tasks to make progress at steady rates that are proportional to their shares. Its strict eligibility control makes the algorithm not work conserving (i.e., a processor may stay idle while there exist runnable tasks in the system that could be executed).

Soon PD was extended by its authors to support dynamic join/leave operations while preserving P-fairness for the single processor case [16]. Anderson and Srinivasan presented PD^2 [120], a work conserving variant of PD (also referred to as Early-relase Fair Scheduling). Furthermore, they analyzed several proportionate fair algorithms and derived the sufficient conditions for supporting dynamic join/leave operations on multiple processors [121]. Finally, Deadline Fair Scheduling (DFS) [35] relaxes the notion of strict P-fairness making it more practical in real systems. DFS allows processors to schedule tasks at different points and to use variable scheduling quanta. It supports dynamic join/leave operations, preserves the work conserving property, and also takes processor affinity into account during scheduling.

13.3.2.3 Other proportional share schedulers

Another main concern of scheduling algorithms is the scheduling overhead, which determines the scalability of the system. If the scheduling overhead is too high, the system makes very little progress in the presence of a large number of runnable tasks. The scheduling overhead for most fair queueing and proportionate fair schedulers is $O(\log n)$ due to the fact that they need to maintain one or more sorted queues. Recently, several CPU schedulers have been developed to provide proportional share with constant overhead.

Virtual Time Round Robin [102] developed by Nieh et al. combines the virtual time concept used by fair queueing algorithms with a round robin scheduling mechanism. The scheduling overhead of VTRR is mostly $O(1)$ except for resetting counters, which takes $O(n)$ time but happens infrequently. Empirical results in this work showed that VTRR's allocation error is comparable to that of WFQ. Group Ratio Round Robin (GRRR) [32] extends VTRR to support scheduling on multiple processors. Trio [82] extends the existing Linux scheduler (the $O(1)$ multi-level queue scheduler used before Linux switched to the new fair queueing based CFS scheduler) to support precise proportional share. It uses an algorithm called distributed weighted round robin (DWRR) to achieve a constant allocation error bound while preserving the $O(1)$ scheduling overhead.

13.3.2.4 Summary

Proportional share schedulers provide more accurate proportional fairness, and some of them even bound the allocation error within a constant. This forms the necessary condition for supporting response-time sensitive applications. In a static system, (where the set of runnable tasks does not change), ensuring the responsive time guarantee of a task becomes the problem of choosing the proper weight for it. This is not as easy as it may seem, because choosing a set of weights to satisfy all tasks may not even be possible. Other than that, constant allocation error is still far from sufficient for ensuring response time guarantees, especially for general purpose operating systems, in which no admission control is conducted and applications may enter/leave at any time.

The critical issue of proportional share schedulers is that they do not provide the necessary performance isolation among tasks. The actual bandwidth received by each task is relative. The more tasks that are running, the less bandwidth each of them receives. Even changing the weight of a task could affect the performance guarantees of other running tasks. As the load in the system increases, response-time sensitive tasks will eventually fail to retain the bandwidth they need. Hierarchical schedulers such as SFQ [61] partially solve the problem by dividing tasks into classes and manage them hierarchically. These schedulers ensure the performance isolation among different classes only as long as the class hierarchy or weight assignment stays fixed. Also, as shown in [109], determining the optimal weight assignment for different classes and isolating tasks within a class still remain challenging problems.

13.3.3 Supporting (Soft) Real Time applications

The most important two things for satisfying the time constraints of (soft) real time applications in general purpose operating systems are: 1) reducing their scheduling latency especially for those scheduling algorithms that do not have constant lag; 2) allocating stable CPU bandwidth to them even if the system is overloaded, which most proportional share algorithms are not capable of. Several schedulers have been proposed to offer better support for response time sensitive applications by extending existing proportional share algorithms. Mechanisms used include dynamically adjusting the weight assignment, modifying virtual times, and adopting the EDF algorithm.

A-SFQ [109] is an adaptive variant of the hierarchical SFQ scheduler, which dynamically changes the weight assignment of application classes to maintain stable CPU bandwidth for the class serving soft real time applications. A-SFQ focuses on two application classes, multimedia (MM) and best-effort (BE). It maintains one performance index for each application class: the percentage of missed deadlines for the MM class and the total slowdown for the BE class. At the end of each time interval, the value of these performance indices are calculated. If the MM class misses too many deadlines, A-SFQ increases the proportion of CPU bandwidth allocated to it by a *chunk* (5%). Changing the proportion allocated to a class is done by re-computing the weights of all classes. The adaptive algorithm used by A-SFQ works quite well for periodic tasks due to their regular execution pattern, but is not suitable for a large set of co-existing aperiodic tasks. Furthermore, it does not provide performance isolation for tasks within a class, since there is nothing prevent users from running too many tasks in the MM class.

Borrowed Virtual Time (BVT) scheduling, proposed by Duda and Cheriton [53], puts its focus on improving response time guarantees in general purpose operating systems. It performs the basic scheduling using a start time driven fair queueing algorithm, which is very similar to SFQ (in fact, it is exactly the same algorithm used by the latest CFS scheduler in Linux). However, such a scheduler is proven to have potentially $O(n)$ lag, which means the work of a response-time sensitive task can be delayed significantly by other tasks that have the same or smaller virtual time. In order to solve this problem, BVT introduces the concept of *wrap*. A response-time sensitive task is allowed to wrap back in virtual time by a negative value to make it appear earlier and thereby gain dispatch preference. BVT effectively schedules those tasks with non-zero *wrap* earlier without changing their proportional share. Apparently, if the system is heavily overloaded, response-time sensitive tasks still cannot obtain their desired CPU bandwidth due to BVT's proportional share nature. The authors claimed that BVT can be configured to support a fixed set of real time applications whose CPU requirements are known in advance, or a dynamic set of best-effort tasks, or a mix of both using two level scheduling. However, the problem of how to support a large dynamic set of response-time sensitive tasks with unpredictable CPU loads is not addressed.

BERT [18] is a scheduler that designed to run a mix of best-effort and real time tasks using a modified WFQ algorithm. In addition to classifying tasks into best-effort and real time, it also divides tasks into important and unimportant based on user's choices. It allows an important real time task to *steal* CPU cycles from unimportant tasks dynamically. BVT attempts to predict the length of each CPU request from real time tasks. If the current share assigned to a real time task is not sufficient for it to meet the deadline, BVT picks one or more unimportant tasks and steals cycles from them by modifying their virtual deadlines (VFT). Suppose, an important real time task P_i with weight w_i needs to steal cycles from another task P_j with weight w_j. During the interval of stealing, P_i has weight $w_i + w_j$ and P_j has weight 0. Their VFT are then adjusted using:

$$\text{new } VFT(P_i) = \text{old } VFT(P_i) - \varepsilon \times w_j / w_i$$

$$\text{new } VFT(P_j) = \text{old } VFT(p_j) + \varepsilon$$

where ε is the duration of the stealing interval, which is determined using the work length and deadline of the stealing task, as well as the weights of all tasks involved. BVT effectively sacrifices unimportant tasks to maintain the CPU bandwidth needed by real time tasks. However, it applies only to well-behaved periodic tasks whose CPU request length is predictable and whose deadline is known, and there is no admission control for preventing the system from being overloaded by too many real time tasks.

SMART [99, 100, 101] is yet another WFQ based scheduler designed to run a mix of best-effort and real time tasks. It explicitly decouples the importance and response time constraints. SMART provides a common importance attribute for both real-time and best-effort tasks based on priorities and virtual finish times as defined in WFQ. SMART then uses an earliest deadline based planner to optimize the order in which tasks are serviced to allow real-time tasks to meet their time constraints. In the face of CPU overloading, tasks of lower importance are sacrificed. In addition, the biased virtual finish time (BVFT) for best-effort tasks that accounts for their ability to tolerate more varied service latencies is used to give interactive and real-time tasks better performance during periods of transient overload. SMART requires applications to supply an estimate of their service time and deadlines so that its execution planner can optimize the service order. The worst case complexity for the execution planner to find a feasible schedule is $O(N^2)$ where N is the number of active real time tasks. Therefore, the requirement of precise a priori application information and high scheduling overhead limit its applicability.

BEST [11] extends a traditional time sharing scheduler with an earliest deadline first mechanism to support periodic soft real time tasks directly. It detects the period of a periodic task, p, by monitoring the intervals between the time when the task enters the run queue. When a periodic tasks enters the run queue at time t, BEST sets its deadline to be $t+p$. For non periodic tasks, BEST uses a value larger than a predefined threshold, $p_{max} = 5.12$ seconds, so that their deadlines appear later than those of periodic tasks. BEST picks the task with the earliest deadline for execution. Once selected, a task is given a time slice, which determines the length it entitled to execute. The time slice for a periodic task is the detected period and the time slice for a non-periodic task is determined by its *nice* value. A larger *nice* value yields a larger time slice, which gives a reasonable proportional share over long intervals. BEST has several issues that limit its practicability. First, it works only for periodic tasks whose period is smaller than a predefined threshold. Second, periodic tasks can easily monopolize the CPU due to the time slice assignment. Therefore, BEST can support only a few well behaved periodic tasks that are carefully selected by the user. Finally, BEST does not actually isolate periodic tasks from other tasks. Here is an example: Suppose, the period used for non-periodic tasks is $p_{non} = 10$ seconds and the time slice is $ts = 10$ms. Then launching $p_{non}/ts = 1000$ non-periodic tasks is enough to cause all periodic tasks miss their deadlines. In other words, a simple fork bomb is bad enough to make the whole system non-responsive.

These schedulers arose as a result of the need to support multimedia applications in general purpose systems. Among them, SMART is the best that uses a dynamic load control to satisfy as many real time applications as possible. Most of the schedulers are specifically designed to support periodic applications by exploiting their regular execution pattern. This means that they cannot efficiently handle a dynamic set of co-existing aperiodic applications. Due to the lack of proper admission control, which is a crucial component for any practical implementation of such a system, the user is responsible for selecting a limited number of well behaved periodic applications and supplying necessary application information, such as the estimated work length and deadlines.

13.4 CPU Scheduling for (Soft) Real Time Systems

When the set of applications and their periods and worst-case execution times are known in advance, real-time scheduling can be done using static priority, such as rate monotonic scheduling, which executes the task with the shortest period first. Most general purpose operating systems, such as Linux and Windows, extend their time-sharing schedulers to support a real-time scheduling class that has strictly higher priorities than any best-effort tasks. Tasks running in this class alway get executed first and preempt best-effort tasks. Without regulating the execution of real-time applications, these schedulers suffer a well-known pathology: starvation of best-effort applications.

Soft real-time scheduling servers [38, 83] take advantage of such a structure. The authors proposed to run a user-level scheduling server at the highest priority in the real-time scheduling class. The scheduling server computes a schedule based using a table-driven CPU reservation algorithm, and runs real-time applications at other priorities. BeOS [95], positioned as a multimedia desktop operating system, explicitly ties the task latency requirements to static priorities, with shorter latencies mapping to higher priorities.

Although static priority schedulers are simple and efficient, they fail to isolate applications from one another if necessary enforcement mechanism are not implemented. Finding the optimal priority assignment requires coordination across application developers, which further makes the approach not suitable for an open desktop environment where the set of applications changes dynamically.

13.4.1 Bandwidth/Execution Rate Control

Jeffay proposed a system based on EDF for serving [71]. At the creation of each request, its worst-case execution time (WCET) and minimal inter-arrival time (MIT) are decided and then the scheduler uses EDF. While WCET has an upper bound for a periodic task, MIT does not. Jeffay then proposed Rate Based Execution (RBE) [70] to force each task to progress at a fixed rate (based on its specification) towards its deadline, so that requests that arrived early are processed as if they arrived at a later time. An admission test and negotiation of rate were implemented to ensure feasibility.

Deng et al. proposed to support a mix of real-time and non-real-time applications by regulating execution using servers [46], such as the Constant Utilization Sever (CUS) [47], the Total Bandwidth Server (TBS) [117, 119, 118], and the Constant Bandwidth Server (CBS) [1]. Each application is encapsulated in a server. The server then uses the parameters, supplied either by applications or the system administrator, to control execution of the applications running inside it, and thereby, provides isolation across servers. The underlying scheduler allocates CPU time to servers using an EDF-based algorithm. The creation of a server must pass an admission test to prevent the system from being overloaded.

Later on, this server-based structure was extended to support the *uniformly slower processor* (USP) abstraction in PShED [84] and *an open environment for real-time applications* [48]. Such a system guarantees that any application that can be scheduled by a processor of speed S can also be scheduled if it is given an USP with rate S/F on a processor of speed F. Each server restricts the applications running in it to use no more than the assigned bandwidth, and therefore, the system can potentially be not work conserving.

Feedback-controlled EDF (FC-EDF) [87, 86] and SWiFT [122] take feedback from applications and dynamically adjust CPU utilization so that the deadline miss rate falls in a controlled range. In general, they provide no guarantees to individual applications, but they are able to achieve higher system utilization with few deadline misses over all applications. In this sense, Redline's CPU scheduling is similar to them. Redline takes the overall CPU bandwidth consumed by reservations as feedback, and ensures system responsiveness by controlling it within a desired range.

13.4.2 CPU Reservations

The *CPU reservations* mechanism used by Redline allows a task to reserve C units of CPU time in every period of T. For example, a task could reserve 5ms of CPU time out of every 30ms. Its implementation requires an *admission test* to ensure the system is not overloaded, and an enforcement mechanism to prevent tasks from consuming more than reserved.

CPU reservations can be implemented in various ways. Nemesis [81], CPU service class [39], and Coulson et al. [44] use EDF, Linux/RK [103] and Mercer et al. [93, 94] use RM, the scheduler by Lin et al. [83] is table driven, and Rialto [75] assigns CPU time intervals using a tree-based data structure.

CPU reservations are capable of support co-existing, possibly misbehaving applications if an appropriate enforcement mechanism is implemented. They eliminate the need for global coordination, since application resource requirements are stated in *absolute* units rather than *relative* units like priority or share. However, unlike Redline, these schedulers perform their admission test based strictly on application specifications without taking the actual load into account. Such admission tests are too pessimistic, seriously limiting the number of co-existing application that a system can support. Consequently, they are not suitable for handling a large dynamic set of response-time sensitive applications (both periodic and aperiodic) that a modern general purpose operating system must support.

13.5 CPU Scheduling Frameworks

Due to the richness of applications that run on modern computer systems, a single CPU scheduling algorithm is not enough to handle the diverse requirements of these applications. Most systems now offer scheduling frameworks that allow multiple scheduling policies to co-exist. A commonly used approach is to organize schedulers in a hierarchical manner. The *root* scheduler allocates CPU time to schedulers below it in the hierarchy, which in turn allocate CPU time to their children, and so on until a leaf is reached.

13.5.1 Two-Level Scheduling Frameworks:

The Linux and Windows schedulers are conceptually hierarchical in the sense that the root scheduler always allocates CPU time first to the real time scheduler and then to the time-sharing scheduler. Redline exploits such a framework and extends the system with an EDF based scheduler that is dedicated to serve interactive applications.

Nemesis [81] uses a two-level scheduling hierarchy with an EDF based root scheduler inside the kernel and second-level schedulers in user space. *Scheduler activations* [7] follows the same architecture, allowing the kernel to notify user-level thread schedulers of certain events that may affect their scheduling decisions, so that they can react accordingly. The Spin operating system [22] implements a two-level hierarchy inside the kernel that has functionality similar to scheduler activations. It has a fixed root scheduler, but allows applications to load their own second-level schedulers into the kernel at run time. Vassal [31] is another system that provides support for a dynamically loadable scheduler, allowing applications to select suitable scheduling policies. However, only a single scheduler can be loaded at a time, in addition to the standard time-sharing scheduler.

RED-Linux [132] proposes a two-level scheduling framework in which tasks are scheduled based on *effective priority*, produced from the task's scheduling parameters: priority, start time, finish time and budget. Each task must belong to one of the leaf schedulers, which may implement different policies. The root scheduler always picks the task with the highest effective priority and lets the leaf scheduler execute the task using its specific policy.

The scheduling frameworks developed in PShED [84] and by Deng et al. [48] have an EDF based root scheduler and various *servers* as its leaf schedulers. The ideas is to provide a *uniformly slower processor* (USP) abstraction to each real time application. This abstraction ensures that each application will receive a given share of the processor bandwidth as if it is running on a slower processor. FSF [2] proposed by Aldea et al. goes further to allow applications to supply requirements using the *service contracts*, instead of directly using server scheduling parameters. Based on the contract of an application, the root scheduler verifies the resources needed and creates a suitable server that encapsulates it. This isolates requirement specification from the detailed system implementation. Therefore FSF supports various scheduling policies for both the root and leaf schedulers.

13.5.2 Multi-Level Scheduling Frameworks:

Stride [131], proposed by Waldspurger, is a hierarchical fair queueing based scheduler. It groups the clients hierarchically into a binary tree, and recursively applies the basic stride scheduling algorithm at each level. BVT [53] can be configured to a hierarchical fashion such that part of the processor bandwidth is reserved for multimedia applications and best-effort applications are scheduled in a time-sharing manner using a second-level scheduler.

These scheduling frameworks are considered to be homogeneous in the sense that all nodes in the hierarchy use the same scheduling algorithm.

Goyal et al. developed a proportional share scheduling algorithm, *start-time fair queueing* (SFQ) [61], and proposed a hierarchical architecture on top of it. In this scheduling hierarchy, all internal points use SFQ schedulers, and leaf nodes may use arbitrary scheduling algorithms, such as static-priority scheduling or rate monotonic. Such a hierarchical architecture allows users to group various applications into classes and to achieve quite strong isolation among classes as long as the scheduling hierarchy stays unchanged. However, no method for providing isolation for applications running in the class was presented.

CPU Inheritance Scheduling [58] permits any task to act as a scheduler by *donating* the CPU to other tasks. The root of the scheduling hierarchy is a special task that the operating system always donates the CPU to after the occurrence of a scheduling event. The scheduling hierarchy exists as a set of informal relationships between tasks, requiring very little support from the kernel. However, it stipulates that all composition issues are the responsibility of the user. The Exokernel [55] uses a CPU donation primitive similar to the one used in CPU inheritance scheduling. The difference is that the root scheduler for Exokernel is a fixed time-sharing scheduler.

The Hierarchical Loadable Scheduler (HLS) [110, 111] is a powerful heterogeneous framework that allows user-specified scheduling algorithms to exist everywhere in the hierarchy. HLS is capable of providing a collection of hierarchical schedulers, allowing it to match each application request with a scheduler that can meet its requirement. It allows various scheduling policies, as well as combinations of them, to be loaded dynamically into the operating system to meet the scheduling requirements of a particular usage scenario. Since the authors focused on the abstraction and interface needed for composition of schedulers, they did not discuss the detailed admission control tied to each specific scheduling algorithm.

13.6 Memory Management

In this section, we describe the related work in the area of memory management. Specifically, we focus on the page replacement mechanisms that are used to resolve memory contention, starting from general purpose operating systems.

13.6.1 Page replacement in General Purpose Systems

Despite the many replacement algorithm proposed, approximations of Least Recently Used (LRU) replacement are predominant in actual systems because of its simplicity and efficiency.

The Linux VMM uses a page replacement algorithm that approximates a global LRU policy, in which pages of all processes are kept in approximate LRU order in a set of global queues. Pages used longest ago are cleaned and selected for reclamation. The Windows VMM adopts a per-process working set model. It uses a kernel thread to steal least recently used pages away from processes and move them into a *standby* list. When there is not enough memory, pages in the *standby* list are reclaimed. Neither Linux nor Windows has the ability to protect the working sets of interactive applications under heavy memory contention, due to their relatively low memory reference rate.

Token-ordered LRU [72] allows *one* task in the system to hold the token for a period of time and build up its working set. By default, each task is allowed to hold the token for 300 seconds and then passes it to the next task. Given the complex interactions among applications, a single winner is far from enough to keep the system responsive under memory pressure. For example, most GUI applications rely on the X server, desktop/window manager, and various communication daemons to work properly. Furthermore, in multiprocessor systems, this mechanism leads to low CPU utilization.

In Zhou et al. [138], the VMM maintains a miss ratio curve for each process, whose pages are kept in LRU order. Such a curve gives an estimate of the rate of page faults a process would incur given a memory allocation size k. Using this information, the VMM reclaims pages from the process that incurs the least penalty (i.e., triggers fewest page faults in the future) in the face of memory pressure. The approach is still throughput oriented, and oblivious to response time requirements.

EELRU [116] is an adaptive algorithm that uses a simple on-line cost/benefit analysis to guide replacement decisions. Compressed Caching [133] uses part of the available memory to hold pages in compressed form, bridging the huge performance gap between memory and disk. Informed prefetching [106, 76] attempts to pre-load pages that are likely to be used soon based on the application's reference behavior. However, none of them addresses the problem of preventing pages used by interactive applications from being evicted in the first place.

13.6.2 Application Specific Paging

The user-level external pager in the Mach operating system was originally designed for writing back evicted pages to disk using application-supplied methods. It was later extended to support application-specific paging [92, 65, 79]. These systems allow applications to change the operating system's replacement decisions by providing alternative pages based on their own page replacement policies. However, they do not have the ability to prevent the underlying operating system from reducing an application's memory allocation.

ExOS is an extensible library operating system that runs on top of a micro-kernel, the ExoKernel [55]. ExOS manages most fundamental operating system abstractions at the application level (i.e., in user space). It provides a rudimentary virtual memory system, AVM [54], that supports application-specific operations. However, page swapping is not supported. Nemesis [81], also based on a micro-kernel architecture, pushes most resource management into user space. Nemesis allocates physical memory to applications according to their contracts. Applications are allowed to manage allocated physical memory based on their own policies using a technique called *self-paging* [64].

SPIN [22] is an extensible operating system that can be dynamically specialized using event-driven extensions. When SPIN's physical page service needs to reclaim a page, it raises a *recall* event on that page. Upon receiving the event, the application specific handler can then volunteer an alternative page based on its own policy. Unlike the systems described above, SPIN loads and executes application supplied extensions in kernel address space, rather than in user address space.

In the Bookmarking GC [67], the VMM informs the garbage collector that it is about to evict a particular page. The garbage collector then processes the information on that page and bookmarks it, so that it is not referenced during later garbage collections. The bookmarking GC is also capable of telling the VMM to evict an alternative page that has no useful data.

13.6.3 Tracking the Working Set

The CRAMM VM computes *stack distances*, which were originally designed for trace analysis. Mattson et al. introduced a one-pass algorithm, based on stack distances, that analyzes a reference trace and produces cache misses for caches of any size [89]. This algorithm was later adapted by Kim and Hsu to handle highly-associative caches [78]. However, these algorithms compute stack distance in linear time, making them too slow to use inside a kernel. Subsequent work on analyzing reference traces used more advanced dictionary data structures [4, 20, 51, 104, 125]. These algorithms calculate stack distance in logarithmic time, but do not maintain underlying referenced blocks in order. This order is unnecessary for trace processing but crucial for page eviction decisions. The CRAMM VM maintains pages in a list that preserves potential eviction order, and uses a separate AVL tree to calculate stack distance in logarithmic time. Transparent contribution of memory [41] uses a similar LRU histogram-based approach to calculate the working sets of applications in user space.

Zhou et al. present a VM system that also tracks LRU reference curves inside the kernel [138]. They use Kim and Hsu's linear-time algorithm to maintain LRU order and calculate stack distances. To achieve reasonable efficiency, this algorithm requires the use of large group sizes (e.g., 1024 pages), which significantly degrades accuracy. They also use a static division between the active and inactive lists, yielding an overhead of 7 to 10%. The CRAMM VM not only computes stack distance in logarithmic time, but it also can track reference histograms at arbitrary granularities. Furthermore, its size adjustment algorithm for the inactive list allows it to collect information accurately from the tail of miss curves while limiting reference histogram collection overhead to 1%.

13.7 Disk I/O Management

Disk I/O management in modern operating systems stretches through several different layers. 1) The disk cache layer buffers file data in memory to improve file access time, which technically belongs to memory management. It has to determine when cached dirty data should be written back to disk. 2) The file system layer handles how file data are organized and located on the disk. Its journalling system has responsibility to carry out file system updates along with their corresponding data such that the consistency of the file system is not compromised. 3) The device layer interacts with disk devices to perform actual I/O operations. It transforms the file data into appropriate format, and uses an I/O scheduler to serve various disk I/O requests.

In order to preserve application responsiveness, all these layers must coordinate with each other, as well as other resource managers in the system, to respect CPU scheduling goals. Unfortunately, these components are implemented to work independently without such necessary coordination in mind, and consequently are completely oblivious to response time requirements of applications. The lack of coordination can undermine interactivity. For example, the journaling file system can become stuck in committing a large transaction, blocking interactive tasks for a period of time, and the I/O scheduler may serve requests from best-effort tasks before those of interactive tasks, imposing unnecessary delay on them.

As the demand for supporting real-time tasks increased, researchers put much of their effort on I/O device management, and proposed a significant number of I/O scheduling algorithms for real-time and multimedia operating systems. However, there was almost no attention paid to the disk cache and file system layers, since they are considered irrelevant to responsiveness. As a result, the related work presented in this section focuses mainly on the device layer, in particular, the I/O scheduling frameworks and algorithms.

13.7.1 I/O Scheduling in General Purpose Systems

Most general purpose systems handle disk I/O requests using the traditional SCAN algorithm, which is also referred as the *elevator* algorithm, or its variants, such as C-SCAN, LOOK, and C-LOOK [115]. The SCAN algorithm sorts requests in the *dispatch queue* according to their physical block locations on the disk, and services them in sorted order to avoid unnecessarily long seeks. SCAN is designed to maximize overall throughput, but does not take timing constraints into account. Both FreeBSD and Solaris provide SCAN as the default I/O scheduling policy to block device drivers, while allowing them to implement their own.

Linux uses a more flexible *elevator* based I/O scheduling framework. It allows disk devices to choose different policies for determining when I/O requests are moved into the *dispatch queue* and served in SCAN order. Currently Linux provides four I/O scheduling policies that can be dynamically changed: 1) **Noop**: I/O requests are directly moved to the dispatch queue and served using the SCAN algorithm. 2) **Deadline**: each I/O request is associated with a deadline. Once the deadline is passed, the request is moved into the dispatch queue regardless of its physical location to avoid starving. 3) **Anticipatory** [68]: It implements the algorithm proposed by Iyer et al., which further improves throughput by overcoming the deceptive idleness between I/O requests. It pauses for a short time (a few milliseconds) after a read operation in *anticipation* of another read request close by. 4) **CFQ**: I/O requests are moved into the dispatch queue according to their class (e.g., real-time, best-effort, and idle) and priorities. It allows the user to set an I/O priority for each application through a system call (a privileged service). However, it allows I/O requests from the real-time class to starve requests from other classes, and does not distinguish requests coming from interactive and best-effort applications.

The disk I/O scheduling in Windows [112] is priority driven. Each application has an I/O priority in the range of (0–7), which indicates the urgency or importance of its requests. It also allow applications to reserve I/O bandwidth through a special interface to provide better support for multimedia applications.

13.7.2 I/O Scheduling in (Soft) Real Time Systems

Traditional SCAN-based I/O schedulers favor the physical block location of I/O requests over their timeliness, which is a poor fit for real time applications. Therefore, most (soft) real time systems have their own specialized I/O schedulers that take deadline (or latency) constraints into account.

Molano et al. [96] developed an I/O scheduler that uses the resource reservation abstraction in RT-Mach [127]. It allows real-time applications to reserve disk I/O bandwidth, and schedules requests using an EDF algorithm. However, they found this simple mechanism, that focuses solely on timeliness without taking the current disk head position into account, can waste a lot of time on disk rotation and seeking, degrading throughput dramatically. They then improved the algorithm by allowing lower priority requests closer to the current disk head position to be served as long as they do not cause higher priority requests to miss their deadlines. They named the algorithm Just In Time Disk Scheduling (JIT).

Deadline Sensitive SCAN (DS-SCAN) [60] adopts a similar approach using a simpler implementation. DS-SCAN places each I/O request in two queues: one queue ordered by scan position, and another queue ordered by *start deadline*, which indicates the latest time a request must be issued to avoid missing its deadline. It services the next request in the SCAN ordered queue only if this would not cause the first request in the other queue to miss its deadline. Otherwise it services the request with the earliest *start deadline*. Both JIT and DS-SCAN require the scheduler to have reasonably accurate estimates of disk parameters, such as the disk head location and seek time.

Nemesis [81] has an I/O scheduler that provides rate guarantees to real time applications (e.g., continuous media), called RSCAN [14]. In Nemesis, applications perform I/O by sending read or write requests to the *Rbufs* channels. RSCAN uses a credit scheme based on “leaky buckets” to control the rate at which the requests of each channel are added to the underlying *dispatch queue*. The *dispatch queue* is serviced using the classical SCAN algorithm. The bucket size and rate are translated from high level specifications of applications, which have to pass a pessimistic admission test. The remaining slack time in the system is evenly distributed to channels with pending requests as extra credits.

The multimedia operating system QLinux has an integrated I/O scheduler, Cello [113], which also adopts the “leaky buckets” plus SCAN scheme. QLinux divides applications into classes and manages resources for them using hierarchical proportional share. Therefore, in Cello, the leaky bucket parameters (e.g., the bucket size and rate) are translated from the weight of each class, as opposed to the application specifications in RSCAN. Consequently, the bandwidth provided by Cello is relative, while RSCAN is absolute.

13.8 Multiple Resource Management in Operating Systems

In this section, we discuss the approaches to integrated management of multiple resources in operating systems. They fall into two main categories, depending on their designed purposes: (1) providing coarse granularity performance isolation by partitioning resources, and (2) supporting (soft) real time applications by managing/reserving resources needed by individual applications. Table 13.2 presents a brief comparison between Redline and several representative operating systems discussed in this section.

	Admission	**Performance isolation**		**Intg. Mgmt.**		**Without**
	control	**inter-class**	**intra-class**	**mem**	**I/O**	**app. mod.**
Linux, Windows						√
Eclipse [29]		strong		√	√	√
SPU [129]		strong		√	√	√
Solaris Containers [90]		strong		√	√	√
QLinux [126]		strong			√	√
Linux-SRT [37]	pessimistic	strong			√	
RT-Mach [127]	pessimistic	strict	strict		√	
Rialto [74]	pessimistic	strict	strict		√	
Nemesis [81]	pessimistic	strict	strict		√	
Redline	load based	strong	dynamic	√	√	√

Table 13.2. A comparison of representative operating systems to Redline

13.8.1 Performance Isolation

Eclipse [29] uses *reservation domains* to manage multiple resources (CPU, memory, and disk I/O). Each domain has a contract in which resource requirements are stated as a fraction (0%–100%) of the total amount available. Extra resources in the system are evenly distributed to all the domains. Later, Eclipse evolved into Eclipse/BSD [30, 26] which manages CPU, disk, and network bandwidth through hierarchical proportional share. Eclipse/BSD's CPU scheduler uses the MTR-LS (Move-To-Rear List Scheduling) algorithm, and manages the network and disk I/O bandwidth using a fair queueing algorithm. (Eclipse uses weighted round robin for handling disk I/O, but does not handle network bandwidth.) When there is not enough memory in the system, Eclipse/BSD evicts pages from a domain that holds more memory than the amount stated in its contract.

Verghese et al. proposed an approach very similar to Eclipse for supporting performance isolation in shared-memory multiprocessor systems [129]. It manages resources using *software processing units* (SPU), each of which also has a contract indicating its resource requirements. The system attempts to distribute SPUs onto different processors, and uses fair-share scheduling to allocate CPU time among SPUs if they have to share one processor. The amount of memory for each SPU is allocated according to the contract. For disk I/O, the system uses the C-SCAN algorithm (a variant of SCAN). However, in order to control the shares allocated, the system monitors the I/O bandwidth consumed by each SPU. Once an SPU consumes more than its entitled share, it is throttled. By contrast with Eclipse, in which the resource partition is static, an SPU is allowed to borrow resources from another under-loaded SPU and return them back when needed. This gives better sharing and improves overall resource utilization.

Solaris Containers [90] implements light-weight virtual machines within a single operating system instance. It comprises a combination of system resource control and boundary separation provided by *zones*. Zones are configured to act as completely isolated servers, with their own portion of the file system hierarchy and some duplicated service daemons. Each zone can be configured to have a fixed amount of resources. It can receive its desired share via the Solaris fair-share scheduler. Its physical memory allocation can be controlled by the resource capping daemon. However, Solaris containers do not specify how disk I/O is handled.

These systems are designed to provide coarse granularity isolation among high-level entities, such as users and application groups, but not to provide response time guarantees for individual applications. They partition the available resources in the system, and achieve performance isolation by protecting the resource boundaries among partitions. However, they all leave unspecified how resources should be allocated to applications within a partition. In essence, they are orthogonal to Redline, which ensures the responsiveness of *individual* applications using integrated resource management.

13.8.2 Supporting (Soft) Real-Time Applications

In order to support time critical applications, a real-time system has to schedule multiple resources, such as CPU, disk I/O, and network bandwidth, so that necessary services are delivered in a timely fashion.

QLinux [126] is a multimedia operating system that provides integrated resource management for CPU, disk I/O, and network bandwidth using hierarchical proportional share.

The whole system is driven by weights. It allocates CPU and network bandwidth using the hierarchical SFQ algorithm and allocates disk I/O bandwidth using a rate-controlled scheduler, Cello [113]. It requires the system administrator to partition the applications carefully into classes, configure the hierarchy, and assign proper weights to each class as well as individual applications. Changing the hierarchy or weight assignment can easily undermine the performance guarantees provided.

Linux-SRT [37] is yet another soft real-time system built on Linux. It extends the standard Linux with an new scheduling class, SCHED_QOS, and a corresponding disk I/O scheduling class, IO_QOS. Linux-SRT handles soft real-time applications using a rate monotonic algorithm, and its disk I/O scheduler is priority-driven. However, the detailed implementation of Linux-SRT was not revealed in the paper.

The Integrated Real-time Resource Scheduler (IRS) [60] is a system that performs coordinated allocation and scheduling of multiple resources (CPU, disk, and network) for *periodic* soft real-time applications. It has a resource planner that uses a heuristic multi-resource allocation algorithm to reserve resources needed by applications. In IRS, each application must be completely self resource aware, so that its planner can heuristically calculate a schedule for all the resources involved. In particular, every application has to specify a period, the amount required for each resource, and a *task precedence graph* (TPG) presenting the ordering among the uses of resources. Such a complicated model makes IRS too restrictive for general purpose operating systems.

Resource containers [12] are an operating system abstraction proposed by Banga et al. to support resource management in server systems. All resources needed for an activity (e.g., a process with multiple threads, or processes in a chain) are abstracted into a container. Applications must have a high degree of knowledge about the resources they need, which might not be feasible for some interactive applications. For example, a web browser may not know the amount of memory needed to display a web page. Alicherry et al. developed RCLinux [3], which manages CPU scheduling using resource containers in Linux.

RT-Mach attempts to support predictable real-time scheduler and provide a uniform interface to both real-time and non-real-time applications. It uses a time-driven scheduler based on a rate monotonic scheduling paradigm. Later, Molano et al. extended it with a real-time file system along with a disk I/O scheduler [96].

Rialto [74] is a light-weight, soft real-time kernel, that offers deadline-based scheduling to applications in the form of time constraints. Rialto also provides a modular resource management interface [73] that allows applications to negotiate with the resource planner to reserve needed resources. The resource reservation is represented by a number between zero and one, with one indicating 100% of a particular resource. The Rialto virtual memory system [52] explicitly excludes paging since it "introduces unpredictable latencies", and simply reserves and locks pages used by real-time applications.

Nemesis [81] is a real-time operating system based on a micro-kernel architecture, and thereby pushes the majority of services into user space. It provides a single interface for resource allocation. Each application has a contract containing its specification of resource requirements. The Nemesis CPU scheduler is an EDF-based implementation of CPU reservations, and performs an admission test based on the reservation specification. Nemesis reserves a fixed amount memory to each application according to the contract. Applications have to manage the physical memory allocated to them using an application-level paging technique, called *self-paging* [64]. Its disk I/O scheduler, RSCAN [14], combines the traditional SCAN algorithm with a "leaky buckets"-based rate control, whose parameters are derived from application contracts.

Designed for supporting real-time applications, these systems rely on pessimistic admission tests that are based solely on application specifications. Such admission tests ensure that accepted applications never miss any of their deadlines, but sacrifice resource utilization significantly, limiting their applicability in general purpose operating systems.

Unlike Redline, none of these systems has true integrated resource management. They either lock pages used by real-time applications in memory, or push the burden to applications by allocating (or reserving) a fixed amount memory to them. Furthermore, these systems usually require programmer intervention or sophisticated configuration through special privileged interfaces. This prevents normal users from conveniently using the services, making them less attractive in practice.

CHAPTER 14

CONCLUSION

This chapter concludes the dissertation by summarizing its contributions and discussing avenues for future work.

14.1 Summary of contributions

The emergence of new kinds of applications as well as their diverse needs makes system resource management more challenging than ever before. Our experiences show that coordinating resource managers, both inside and outside the operating system, prevents them from working at cross purposes, and dramatically improve application performance in the face of heavy resource contention.

Supporting garbage-collected applications requires cooperation between operating system and application run-times (i.e., JVMs). We present CRAMM, a new system that enables cooperation between the virtual memory manager and garbage collector. It combines a new virtual memory system with a garbage-collector-neutral, analytic heap sizing model to dynamically adjust heap sizes. In exchange for modest overhead (around 1-2.5% on average), CRAMM improves performance dramatically by making full use of memory without incurring paging. CRAMM allows garbage-collected applications to run with a nearly-optimal heap size in the absence of memory pressure, and adapts quickly to dynamic memory pressure changes, avoiding paging while providing high CPU utilization.

Supporting interactive applications requires cooperation between different subsystems in operating system (i.e., CPU, memory, and disk I/O). We present Redline, a system designed for highly interactive environments. Redline combines lightweight specifications with integrated management of memory, disk I/O, and CPU resources, and uses an adaptive mechanism based on actual workloads to control the applications running in the system. By orchestrating multiple resources, Redline is able to deliver responsiveness to interactive applications even when the system is extremely overloaded.

Finally, we show how server applications can also leverage the support from the Redline system, as an example of cooperation between operating system and applications. The Redline server coordinates its request processing with the underlying Redline's CPU scheduler. As a result, it is able to maximize responsiveness (i.e., the number of requests meeting the desired response time requirement) without sacrificing throughput.

14.2 Future Work

Currently, the memory protection mechanism used by Redline is quite simple, and can easily over-estimate or under-estimate an application's working set. One solution would be to integrate the CRAMM VM into Redline, because it provides more precise information about working sets. This would require some performance tuning of the CRAMM VM, so that it can properly handle the fast phase changes of interactive applications (e.g., GUI interfaces, web browsers).

Redline integrates CPU, memory, and disk I/O management. One future project would be adding network bandwidth management into the integration, which would allow Redline to offer better handling of network applications like web browsers. One challenge is that the network traffic of some interactive applications may not be response-time sensitive. For example, the GUI part of a download manager should be considered interactive, while its download network traffic should not.

BIBLIOGRAPHY

[1] Abeni, Luca, and Buttazzo, Giorgio C. Integrating multimedia applications in hard real-time systems. In *Proceedings of the* 19^{th} *IEEE Real-Time Systems Symposium* (1998), pp. 4–13.

[2] Aldea, M., Bernat, Guillem, Broster, Ian, Burns, Alan, Dobrin, Radu, Drake, José M., Fohler, Gerhard, Gai, Paolo, Harbour, Michael González, Guidi, Giacomo, Gutiérrez, J. Javier, Lennvall, Tomas, Lipari, Giuseppe, Martínez, J. M., Medina, Julio L., Gutiérrez, J. C. Palencia, and Trimarchi, Michael. FSF: A real-time scheduling architecture framework. In *Proceedings of the* 12^{th} *IEEE Real-Time Technology and Applications Symposium* (2006), pp. 113–124.

[3] Alicherry, Mansoor, and Gopinath, K. Predictable management of system resources for Linux. In *Proceedings of the 2001 USENIX Annual Technical Conference, FREENIX Track* (2001), pp. 273–283.

[4] Almasi, George, Cascaval, Calin, and Padua, David A. Calculating stack distances efficiently. In *ACM SIGPLAN Workshop on Memory System Performance* (Berlin, Germany, Oct. 2002), pp. 37–43.

[5] Alonso, Rafael, and Appel, Andrew W. An advisor for flexible working sets. In *Proceedings of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Boulder, CO, May 1990), pp. 153–162.

[6] Alpern, B., Attanasio, C. R., Barton, J. J., Burke, M. G., Cheng, P., Choi, J.-D., Cocchi, A., Fink, S. J., Grove, D., Hind, M., Hummel, S. F., Lieber, D., Litvinov, V., Mergen, M. F., Ngo, T., Sarkar, V., Serrano, M. J., Shepherd, J. C., Smith, S. E., Sreedhar, V. C., Srinivasan, H., and Whaley, J. The Jalepeño virtual machine. *IBM Systems Journal 39*, 1 (Feb. 2000), 211–238.

[7] Anderson, Thomas E., Bershad, Brian N., Lazowska, Edward D., and Levy, Henry M. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst. 10*, 1 (1992), 53–79.

[8] Appel, Andrew. Simple generational garbage collection and fast allocation. *Software Practice & Experiences 19*, 2 (Feb. 1989), 171–183.

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

[10] Baker, Henry G. The Treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices 27*, 3 (March 1992), 66–70.

[11] Banachowski, Scott A., and Brandt, Scott A. Better real-time response for time-share scheduling. In *Proceedings of the* 11^{th} *International Workshop on Parallel & Distributed Real-Time Systems* (2003), p. 124.2.

[12] Banga, Gaurav, Druschel, Peter, and Mogul, Jeffrey C. Resource containers: A new facility for resource management in server systems. In *Proceedings of the* 3^{rd} *USENIX Symposium on Operating Systems Design and Implementation* (1999), pp. 45–58.

[13] Barham, Paul, Dragovic, Boris, Fraser, Keir, Hand, Steven, Harris, Timothy L., Ho, Alex, Neugebauer, Rolf, Pratt, Ian, and Warfield, Andrew. Xen and the art of virtualization. In *Proceedings of* 19^{th} *ACM Symposium on Operating Systems Principles* (2003), pp. 164–177.

[14] Barham, Paul R. A fresh approach to file system quality of service. In *Proceedings of the* 7^{th} *Network and Operating System Support of Digital Audio and Video* (1997), pp. 119–128.

[15] Baruah, Sanjoy K., Cohen, N. K., Plaxton, C. Greg, and Varvel, Donald A. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica 15*, 6 (1996), 600–625.

[16] Baruah, Sanjoy K., Gehrke, Johannes, Plaxton, C. Greg, Stoica, Ion, Abdel-Wahab, Hussein M., and Jeffay, Kevin. Fair on-line scheduling of a dynamic set of tasks on a single resource. *Information Processing Letters 64*, 1 (1997), 43–51.

[17] Baruah, Sanjoy K., Gehrke, Johannes E., and Plaxton, C. Greg. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the* 9^{th} *International Parallel Processing Symposium* (1995), pp. 280–288.

[18] Bavier, Andy, Peterson, Larry, and Mosberger, David. BERT: A scheduler for best effort and realtime tasks. Tech. Rep. TR-587-98, Princeton University, 1999.

[19] BEA WebLogic. Technical white paper JRockit: Java for the enterprise. http://www.bea.com/content/news_events /white_papers/BEA_JRockit_wp.pdf.

[20] Bennett, B. T., and Kruskal, V. J. LRU stack processing. *IBM Journal of R & D 19*, 4 (1975), 353–357.

[21] Bennett, Jon C. R., and Zhang, Hui. WF^2Q: Worst-case fair weighted fair queueing. In *Proceedings of IEEE INFOCOM* (1996), pp. 24–28.

[22] Bershad, Brian N., Savage, Stefan, Pardyak, Przemyslaw, Sirer, Emin Gün, Fiuczynski, Marc E., Becker, David, Chambers, Craig, and Eggers, Susan J. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles* (1995), pp. 267–284.

[23] Blackburn, Stephen M., Cheng, Perry, and McKinley, Kathryn S. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *26th International Conference on Software Engineering* (May 2004), pp. 137–146.

[24] Blackburn, Stephen M., Garner, Robin, Hoffmann, Chris, Khan, Asjad M., McKinley, Kathryn S., Bentzure, Rotem, Diwan, Amer, Feinberg, Daniel, Guyer, Samuel Z., Hosking, Antony, Jump, Maria, Moss, J. Eliot B., Stefanović, Darko, VanDrunen, Thomas, von Dincklage, Daniel, and Wiedermann, Benjamin. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the* 21st *ACM International Conference on Object-Oriented Programming, Systems, Languages and Applications* (2006).

[25] Blackburn, Stephen M., McKinley, Kathryn S., Garner, Robin, Hoffmann, Chris, Khan, Asjad M., Bentzur, Rotem, Diwan, Amer, Feinberg, Daniel, Frampton, Daniel, Guyer, Samuel Z., Hirzel, Martin, Hosking, Antony L., Jump, Maria, Lee, Han, Moss, J. Eliot B., Phansalkar, Aashish, Stefanovic, Darko, VanDrunen, Thomas, von Dincklage, Daniel, and Wiedermann, Ben. Wake up and smell the coffee: evaluation methodology for the 21st century. *Communications of the ACM 51*, 8 (2008), 83–89.

[26] Blanquer, J., Bruno, John L., Gabber, Eran., McShea, M., Özden, B., Silberschatz, Abraham, and Singh, A. Resource management for QoS in Eclipse/BSD. In *Proceedings of the FreeBSD Conference* (Berkeley, CA, 1999), pp. 8–14.

[27] Bovet, Daniel P., and Cesati, Marco. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly, 2005.

[28] Brecht, Tim, Arjomandi, Eshrat, Li, Chang, and Pham, Hang. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications* (Tampa, FL, June 2001), pp. 353–366.

[29] Bruno, John, Gabber, Eran, Özden, Banu, and Silberschatz, Abraham. The Eclipse operating system: Providing quality of service via reservation domains. In *Proceedings of the 1998 USENIX Annual Technical Conference* (1998), pp. 235–246.

[30] Bruno, John L., Brustoloni, José Carlos, Gabber, Eran, Özden, Banu, and Silberschatz, Abraham. Retrofitting quality of service into a time-sharing operating system. In *Proceedings of the 1999USENIX Annual Technical Conference, General Track* (1999), pp. 15–26.

[31] Candea, George, and Jones, Michael B. Vassal: Loadable scheduler support for multi-policy scheduling. In *Proceedings of the* 2nd *USENIX Windows NT Symposium* (Seattle, WA, 1998), pp. 157–166.

[32] Caprita, Bogdan, Chan, Wong Chun, Nieh, Jason, Stein, Clifford, and Zheng, Haoqiang. Group Ratio Round-Robin: O(1) proportional share scheduling for uniprocessor and multiprocessor systems. In *Proceedings of the 2005 USENIX Annual Technical Conference* (2005), pp. 337–352.

[33] Carr, Richard W., and Henessey, John L. WSClock – a simple and effective algorithm for virtual memory management. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles (SOSP)* (Dec. 1981), pp. 87–95.

[34] Chandra, Abhishek, Adler, Micah, Goyal, Pawan, and Shenoy, Prashant. Surplus Fair Scheduling: A Proportional-Share CPU scheduling algorithm for symmetric multiprocessors. In *Proceedings of the* 4^{th} *USENIX Symposium on Operating Systems Design and Implementation* (San Diego, CA, 2000), pp. 45–58.

[35] Chandra, Abhishek, Adler, Micah, and Shenoy, Prashant. Deadline Fair Scheduling: Bridging the theory and practice of proportionate-fair scheduling in multiprocessor servers. In *Proceedings of the* 7^{th} *IEEE Real-Time Technology and Applications Symposium* (2001), pp. 3–14.

[36] Cheung, Shun Y., and Pencea, Corneliu S. BSFQ: Bin-sort fair queueing. In *Proceedings of IEEE INFOCOM* (2002), pp. 1640–1649.

[37] Childs, Stephen, and Ingram, David. The Linux-SRT integrated multimedia operating system: Bringing QoS to the desktop. In *Proceedings of the* 7^{th} *IEEE Real-Time Technology and Applications Symposium* (2001), pp. 135–140.

[38] Chu, Hao Hua, and Nahrstedt, Klara. A soft real time scheduling server in UNIX operating system. In *Proceedings of the* 4^{th} *International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services* (1997), pp. 153–162.

[39] Chu, Hao Hua, and Nahrstedt, Klara. CPU service classes for multimedia applications. In *Proceedings of the* 6^{th} *IEEE International Conference on Multimedia Computing Systems, Vol. 1* (1999), pp. 296–301.

[40] Chu, W. W., and Opderbeck, H. The page fault frequency replacement algorithm. In *AFIPS Conference Proceedings* (Montvale, NJ, 1972), vol. 41(1), AFIPS Press, pp. 597–609.

[41] Cipar, James, Corner, Mark D., and Berger, Emery D. Transparent contribution of memory. In *USENIX Annual Technical Conference, General Track* (2006), pp. 109–114.

[42] Cobb, Jorge Arturo, Gouda, Mohamed G., and El-Nahas, Amal. Time-shift scheduling: Fair scheduling of flows in high speed networks. *IEEE/ACM Trans. Netw. 6*, 3 (1998), 274–285.

[43] Cooper, Eric, Nettles, Scott, and Subramanian, Indira. Improving the performance of SML garbage collection using application-specific virtual memory management. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming* (San Francisco, CA, June 1992), pp. 43–52.

[44] Coulson, Geoff, Campbell, Andrew T., Robin, Philippe, Blair, Gordon S., Papathomas, Michael, and Shepherd, Doug. The design of a QoS-controlled ATM-based communications system in Chorus. *IEEE Journal on Selected Areas in Communications 13*, 4 (1995), 686–699.

[45] Demers, Alan J., Keshav, Srinivasan, and Shenker, Scott. Analysis and simulation of a fair queueing algorithm. In *Proceedings of ACM SIGCOMM* (1989), pp. 1–12.

[46] Deng, Zhong, and Liu, Jane. Scheduling real-time applications in an open environment. In *Proceedings of the* 18^{th} *IEEE Real-Time Systems Symposium* (1997), pp. 308–319.

[47] Deng, Zhong, Liu, Jane, and Sun, J. A scheme for scheduling hard real-time applications in open system environment. In *Proceedings of the* 9^{th} *Euromicro Workshop on Real-Time Systems* (1997), pp. 191–199.

[48] Deng, Zhong, Liu, Jane, Zhang, Lynn Y., Seri, Mouna, and Frei, Alban. An open environment for real-time applications. *Real-Time Systems 16*, 2-3 (1999), 155–185.

[49] Denning, Peter J. The working set model for program behavior. In *Proceedings of the* 1^{st} *ACM Symposium on Operating Systems Principles* (1967), pp. 15.1–15.12.

[50] Denning, Peter J. Working sets past and present. *IEEE Transactions on Software Engineering SE-6(1)* (Jan. 1980), 64–84.

[51] Ding, Chen, and Zhong, Yutao. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, CA, June 2003), pp. 245–257.

[52] Draves, Richard P., Odinak, Gilad, and Cutshall, Scott M. The Rialto virtual memory system. Tech. Rep. MSR-TR-97-04, Microsoft Research, Advanced Technology Division, 1995.

[53] Duda, Kenneth J., and Cheriton, David R. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose schedular. In *Proceedings of the* 17^{th} *ACM Symposium on Operating Systems Principles* (1999), pp. 261–276.

[54] Engler, Dawson R., Gupta, Sandeep K., and Kaashoek, M. Frans. AVM: Application-level virtual memory. In *Proceedings of the HOTOS-V* (1995), pp. 72–77.

[55] Engler, Dawson R., Kaashoek, M. Frans, and O'Toole, James. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles* (1995), pp. 251–266.

[56] Essick, R. An event-based fair share scheduler. In *Proceedings of the Winter USENIX Conference* (1990), pp. 147–162.

[57] Etsion, Yoav, Tsafrir, Dan, and Feitelson, Dror G. Human-centered scheduling of interactive and multimedia applications on a loaded desktop. Tech. Rep. 2003-3, Hebrew University, 2003.

[58] Ford, Bryan, and Susarla, Sai. CPU inheritance scheduling. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation* (1996), pp. 91–105.

[59] Golestani, S. Jamaloddin. A self-clocked fair queueing scheme for broadband applications. In *Proceedings of IEEE INFOCOM* (1994), pp. 636–646.

[60] Gopalan, Kartik, and Chiueh, Tzicker. Multi-resource allocation and scheduing for periodic soft real-time applications. In *Proceedings of the 9th ACM/SPIE Conference on Multimedia Computing and Networking* (Berkeley, CA, 2002), pp. 34–45.

[61] Goyal, Pawan, Guo, Xingang, and Vin, Harrick M. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation* (Seattle, WA, 1996), pp. 107–121.

[62] Grzegorczyk, Chris, Soman, Sunil, Krintz, Chandra, and Wolski, Richard. Isla Vista heap sizing: Using feedback to avoid paging. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization* (2007), pp. 325–340.

[63] Guo, Chuanxiong. SRR: An O(1) time complexity packet scheduler for flows in multi-service packet networks. In *Proceedings of ACM SIGCOMM* (2001), pp. 211–222.

[64] Hand, Steven M. Self-Paging in the Nemesis operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation* (1999), pp. 73–86.

[65] Harty, Kieran, and Cheriton, David R. Application-controlled physical memory using external page-cache management. In *Proceedings of the 5th Interational Conference on Architectural Support for Programming Languages and Operating Systems* (1992), pp. 187–197.

[66] Henry, G. The fair share scheduler. *AT&T Bell Laboratories Technical Journal 63*, 8 (19984), 1845–1857.

[67] Hertz, Matthew, Feng, Yi, and Berger, Emery D. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2005), pp. 143–153.

[68] Iyer, Sitaram, and Druschel, Peter. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (2001), pp. 117–130.

[69] JavaSoft. J2SE 1.5.0 documentation: Garbage collector ergonomics. http://java.sun.com/j2se/1.5.0/docs/guide/vm/ gc-ergonomics.html.

[70] Jeffay, Kevin, and Bennett, David. A rate-based execution abstraction for multimedia computing. In *Proceedings of the* 5^{th} *Network and Operating System Support of Digital Audio and Video* (1995), pp. 64–75.

[71] Jeffay, Kevin, Stone, Donald L., and Smith, F. Donelson. Kernel support for live digital audio and video. *Computer Communications 15*, 6 (1992), 388–395.

[72] Jiang, Song, and Zhang, Xiaodong. Token-ordered LRU: an effective page replacement policy and its implementation in Linux systems. *Perform. Eval. 60*, 1-4 (2005), 5–29.

[73] Jones, Michael B., Leach, Paul J., and Draves, Richard P. Support for user-centric modular real-time resource management in the Rialto operating system. In *Proceedings of the* 5^{th} *Network and Operating System Support of Digital Audio and Video* (1995), pp. 53–63.

[74] Jones, Michael B., McCulley, Daniel L., Forin, Alessandro, Leach, Paul J., Rosu, Daniela, and Roberts, Daniel L. An overview of the Rialto real-time architecture. In *Proceedings of the* 7^{th} *ACM SIGOPS European Workshop* (1996), pp. 249–256.

[75] Jones, Michael B., Rosu, Daniela, and Rosu, Marcel-Catalin. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the* 16^{th} *ACM Symposium on Operating Systems Principles* (1997), pp. 198–211.

[76] Kaplan, Scott F., McGeoch, Lyle A., and Cole, Megan F. Adaptive caching for demand prepaging. In *Proceedings of the 2002 ACM SIGPLAN International Symposium on Memory Management* (2002), pp. 221–232.

[77] Kay, J., and Lauder, P. A fair share scheduler. *Communications of ACM 31*, 1 (1988), 44–55.

[78] Kim, Yul H., Hill, Mark D., and Wood, David A. Implementing stack simulation for highly-associative memories. In *Proceedings of the 1991 SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (San Diego, CA, 1991), pp. 212–213.

[79] Krueger, Keith, Loftesness, David, Vahdat, Amin, and Anderson, Thomas E. Tools for the development of application-specific virtual memory management. In *Proceedings of the* 8^{th} *ACM International Conference on Object-Oriented Programming, Systems, Languages and Applications* (1993), pp. 48–64.

[80] Lee, Jeng Farn, Chen, Meng Chang, and Sun, Yeali. WF^2Q-M: Worst-case fair weighted fair queueing with maximum rate control. *Computer Networks: The International Journal of Computer and Telecommunications Networking 51*, 6 (2007), 1403–1420.

[81] Leslie, Ian M., McAuley, Derek, Black, Richard, Roscoe, Timothy, Barham, Paul, Evers, David, Fairbairns, Robin, and Hyden, Eoin. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications 14*, 7 (1996), 1280–1297.

[82] Li, Tong. http://triosched.sourceforge.net/.

[83] Lin, Chih han, Chu, Hao hua, and Nahrstedt, Klara. A soft real-time scheduling server on the Windows NT. In *Proceedings of the* 2^{nd} *USENIX Windows NT Symposium* (1998), pp. 149–156.

[84] Lipari, Giuseppe, Carpenter, John, and Baruah, Sanjoy K. A framework for achieving inter-application isolation in multiprogrammed hard real-time environments. In *Proceedings of the* 21^{st} *IEEE Real-Time Systems Symposium* (2000), pp. 217–226.

[85] Liu, C. L., and Layland, James W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM 20*, 1 (1973), 46–61.

[86] Lu, Chenyang, Stankovic, John A., Son, Sang Hyuk, and Tao, Gang. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems 23*, 1-2 (2002), 85–126.

[87] Lu, Chenyang, Stankovic, John A., Tao, Gang, and Son, Sang Hyuk. Design and evaluation of a feedback control EDF scheduling algorithm. In *Proceedings of the* 20^{th} *IEEE Real-Time Systems Symposium* (1999), pp. 56–67.

[88] Maheshwari, Umesh. Charge-based proportional scheduling. Tech. Rep. TM-529, MIT Laboratory for CS, 1995.

[89] Mattson, R. L., Gecsei, J., Slutz, D. R., and Traiger, I. L. Evaluation techniques for storage hierarchies. *IBM Systems Journal 9*, 2 (1970), 78 – 117.

[90] Mauro, Jim, and McDougall, Richard. *Solaris Internal: Core Kernel Components*. Sum Microsystems Press, A Prentice Hall Title, 2000.

[91] McKusick, Marshall Kirk, and Neville-Neil, George V. *The Design and Implementation of the FreeBSD Operating System*. Addison Wesley, August 2004.

[92] McNamee, Dylan, and Armstrong, Katherine. Extending the Mach external pager interface to accommodate user-level page replacement policies. In *Proceedings of the 1990 USENIX Mach Symposium* (1990), pp. 17–29.

[93] Mercer, Clifford W., Savage, Stefan, and Tokuda, Hideyuki. Processor capacity reserves: An abstraction for managing processor usage. In *Proceedings of Workshop on Workstation Operating Systems* (1993), pp. 129–134.

[94] Mercer, Clifford W., Savage, Stefan, and Tokuda, Hideyuki. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the* 1^{st} *IEEE International Conference on Multimedia Computing Systems* (1994), pp. 90–99.

[95] Meurillon, Cyril. Be engineering insights: The kernel scheduler and real-time threads. Be Newsletter, 37, August 1996. http://www-classic.be.com/aboutbe/benewsletter/Issue37.html.

[96] Molano, Anastasio, Juvva, Kanaka, and Rajkumar, Ragunathan. Real-time filesystems - guaranteeing timing constraints for disk accesses in RT-Mach. In *Proceedings of the* 18th *IEEE Real-Time Systems Symposium* (1997), pp. 155–165.

[97] Molnar, Ingo. http://people.redhat.com/mingo/cfs-scheduler/.

[98] Moon, David A. Garbage collection in a large LISP system. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (1984), pp. 235–245.

[99] Nieh, Jason, and Lam, Monica S. SMART: A processor scheduler for multimedia applications. In *Proceedings of the* 15th *ACM Symposium on Operating Systems Principles* (1995), p. 233.

[100] Nieh, Jason, and Lam, Monica S. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the* 16th *ACM Symposium on Operating Systems Principles* (Saint-Malo, France, 1997), pp. 184–197.

[101] Nieh, Jason, and Lam, Monica S. A SMART scheduler for multimedia applications. *ACM Trans. Comp. Syst. 21*, 2 (2003), 117–163.

[102] Nieh, Jason, Vaill, Christopher, and Zhong, Hua. Virtual-Time Round-Robin: An O(1) proportional share scheduler. In *Proceedings of the 2001 USENIX Annual Technical Conference* (2001), pp. 245–259.

[103] Oikawa, Shuichi, and Rajkumar, Ragunathan. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the* 5th *IEEE Real-Time Technology and Applications Symposium* (1999), pp. 111–120.

[104] Olken, F. Efficient methods for calculating the success function of fixed space replacement policies. Tech. Rep. LBL-12370, Lawrence Berkeley Laboratory, 1981.

[105] Parekh, Abhay K., and Gallager, Robert G. A generalized processor sharing approach to flow control in integrated services networks: the single node case. In *Proceedings of IEEE INFOCOM* (1992), pp. 915–924.

[106] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. Informed prefetching and caching. In *Proceedings of the* 15th *ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), ACM Press, pp. 79–95.

[107] Prabhakaran, Vijayan, Arpaci-Dusseau, Andrea C., and Arpaci-Dusseau, Remzi H. Analysis and Evolution of Journaling File Systems. In *Proceedings of the 2005 USENIX Annual Technical Conference* (2005), pp. 105–120.

[108] Ramabhadran, Sriram, and Pasquale, Joseph. Stratified round robin: A low complexity packet scheduler with bandwidth fairness and bounded delay. In *Proceedings of ACM SIGCOMM* (2003), pp. 239–250.

[109] Rau, Melissa A., and Smirni, Evgenia. Adaptive CPU scheduling policies for mixed multimedia and best-effort workloads. In *Proceedings of the* 7^{th} *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (Washington, DC, 1999), pp. 252–261.

[110] Regehr, John, and Stankovic, John A. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the* 22^{nd} *IEEE Real-Time Systems Symposium* (2001), pp. 3–14.

[111] Regehr, John David. *Using Hierarchical Scheduling to Support Soft Real-Time Applications in General-Purpose Operating Systems*. PhD thesis, University of Virginia, Charlottesville, VA, USA, 2001. Adviser-John A. Stankovic.

[112] Russinovich, Mark E., and Solomo, David A. *Micrsoft Windows Internals (4th Edition)*. Microsoft Press, 2004.

[113] Shenoy, Prashant J., and Vin, Harrick M. Cello: A disk scheduling framework for next generation operating systems. *Real-Time Systems 22*, 1-2 (2002), 9–48.

[114] Shreedhar, M., and Varghese, George. Efficient fair queueing using deficit round robin. In *Proceedings of ACM SIGCOMM* (1995), pp. 231–242.

[115] Silberschatz, Avi, baer Galvin, Peter, and Gagne, Greg. *Operating System Concepts, Seventh Edition*. John Wiley & Sons, Inc., 2004.

[116] Smaragdakis, Yannis, Kaplan, Scott F., and Wilson, Paul R. The EELRU adaptive replacement algorithm. *Performance Evaluation 53*, 2 (July 2003), 93–123.

[117] Spuri, Marco, and Buttazzo, Giorgio C. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the* 15^{th} *IEEE Real-Time Systems Symposium* (1994), pp. 2–11.

[118] Spuri, Marco, and Buttazzo, Giorgio C. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems 10*, 2 (1996), 179–210.

[119] Spuri, Marco, Buttazzo, Giorgio C., and Sensini, Fabrizio. Robust aperiodic scheduling under dynamic priority systems. In *Proceedings of the* 16^{th} *IEEE Real-Time Systems Symposium* (1995), pp. 210–221.

[120] Srinivasan, Anand, and Anderson, James H. Early-release fair scheduling. In *Proceedings of the* 12^{th} *Euromicro Conference on Real-Time Systems* (2000), pp. 35–43.

[121] Srinivasan, Anand, and Anderson, James H. Fair scheduling of dynamic task systems on multiprocessors. *Journal of System Software 77*, 1 (2005), 67–80.

[122] Steere, David C., Goel, Ashvin, Gruenberg, Joshua, McNamee, Dylan, Pu, Calton, and Walpole, Jonathan. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the* 3^{rd} *USENIX Symposium on Operating Systems Design and Implementation* (1999), pp. 145–158.

[123] Stoica, Ion, and Abdel-Wahab, Hussein. Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation. Tech. Rep. TR-95-22, Old Dominion University, 1995.

[124] Stoica, Ion, Abdel-Wahab, Hussein, Jeffay, Kevin, Baruah, Sanjoy, Gehrke, Johannes, and Plaxton, C. Greg. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the* 17^{th} *IEEE Real-Time Systems Symposium* (Dec. 1996), pp. 288–299.

[125] Sugumar, Rabin A., and Abraham, Santosh G. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Measurement and Modeling of Computer Systems* (Santa Clara, CA, 1993), pp. 24–35.

[126] Sundaram, Vijay, Chandra, Abhishek, Goyal, Pawan, Shenoy, Prashant J., Sahni, Jasleen, and Vin, Harrick M. Application performance in the QLinux multimedia operating system. In *Proceedings of the* 8^{th} *ACM Multimedia* (2000), pp. 127–136.

[127] Tokuda, Hideyuki, Nakajima, Tatsuo, and Rao, Prithvi. Real-time Mach: Towards a predictable real-time system. In *Proceedings of the 1990 USENIX MACH Symposium* (1990), pp. 73–82.

[128] Tsafrir, Dan, Etsion, Yoav, and Feitelson, Dror G. Secretly monopolizing the CPU without superuser privileges. In *Proceedings of the* 16^{th} *USENIX Security Symposium* (2007), pp. 1–18.

[129] Verghese, Ben, Gupta, Anoop, and Rosenblum, Mendel. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *Proceedings of the* 8^{th} *Interational Conference on Architectural Support for Programming Languages and Operating Systems* (1998), pp. 181–192.

[130] Waldspurger, Carl A., and Weihl, William E. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the* 1^{st} *USENIX Symposium on Operating Systems Design and Implementation* (1994), pp. 1–11.

[131] Waldspurger, Carl A., and Weihl, William E. Stride scheduling: Deterministic proportional-share resource management. Tech. Rep. TR-528, MIT Laboratory of CS, 1995.

[132] Wang, Yu-Chung, and Lin, Kwei-Jay. Implementing a general real-time scheduling framework in the RED-Linux real-time kernel. In *Proceedings of the* 20^{th} *IEEE Real-Time Systems Symposium* (1999), pp. 246–255.

[133] Wilson, Paul R., Kaplan, Scott F., and Smaragdakis, Yannis. The case for compressed caching in virtual memory systems. In *Proceedings of the 1999 USENIX Annual Technical Conference* (1999), pp. 101–116.

[134] Yang, Ting, Berger, Emery D., Kaplan, Scott F., and Moss, J. Eliot B. CRAMM: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation* (2006), pp. 103–116.

[135] Yang, Ting, Hertz, Matthew, Berger, Emery D., Kaplan, Scott F., and Moss, J. Eliot B. Automatic heap sizing: Taking real memory into account. In *Proceedings of the 2004 ACM SIGPLAN International Symposium on Memory Management* (Vancouver, Canada, Oct. 2004), pp. 61–72.

[136] Zhang, Chengliang, Kelsey, Kirk, Shen, Xipeng, Ding, Chen, Hertz, Matthew, and Ogihara, Mitsunori. Program-level adaptive memory management. In *Proceedings of the 2004 ACM SIGPLAN International Symposium on Memory Management* (2006), pp. 174–183.

[137] Zhang, Lixia. Virtualclock: A new traffic control algorithm for packet-switched networks. *ACM Trans. Comput. Syst. 9*, 2 (1991), 101–124.

[138] Zhou, Pin, Pandy, Vivek, Sundaresan, Jagadeesan, Raghuraman, Anand, Zhou, Yuanyuan, and Kumar, Sanjeev. Dynamic tracking of page miss ratio curves for memory management. In *Proceedings of the 11th Interational Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, Oct. 2004), pp. 177–188.