RICE UNIVERSITY

Advanced memory management and disk scheduling techniques for general-purpose operating systems

by

Sitaram Iyer

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Peter Druschel, Chairman
Professor
Computer Science

Dan Wallach
Associate Professor
Computer Science

Edward Knightly
Associate Professor
Electrical and Computer Engineering and
Computer Science

Houston, Texas

November, 2005

Advanced memory management and disk scheduling techniques for general-purpose operating systems

Sitaram Iyer

Abstract

Operating systems have evolved into sophisticated, high-performance virtualizing platforms, to support and be fair towards concurrently running applications. However, since applications usually run oblivious of each other and prefer narrow system interfaces, they inadvertently contend for resources, resulting in inappropriate allocations and significant performance degradations. This dissertation identifies and eliminates two such problems: one we call *rigidity in physical memory management* which we solve using *adaptive memory management*, and a second we call *deceptive idleness in disk schedulers* that we solve through *anticipatory disk scheduling*.

Many applications, their libraries, and runtimes can trade memory consumption for performance by maintaining caches, triggering garbage collection, etc. However, due to ignorance of memory pressure in the system, they are forced to be conservative about memory usage. *Adaptive memory management* is a technique that informs applications of the severity of memory pressure via a metric that quantifies the cost of using memory. This enables applications to allocate memory liberally when available

(with performance benefits of 20% to 300%), and to release it under contention. The system thus reaches an equilibrium that balances the impact of memory pressure on each application; adapts to avoid paging during load bursts and improves stability and responsiveness; and reduces the need for manual configuration of memory footprints. It also provides finer control on memory usage by adapting proportional to application priorities.

Disk schedulers generally schedule a request as soon as the previous request has finished. Unfortunately, many applications perform synchronous I/O by issuing a request after the previous request has been served. This causes the scheduler to suffer from *deceptive idleness*, a condition where it incorrectly assumes that the process has no further requests, and seeks to a request from another process. *Anticipatory disk scheduling* transparently solves this problem by sometimes injecting a small, controlled delay into the disk scheduler before it makes a scheduling decision, whenever it expects the current request to be quickly followed by another nearby request. This improves performance by up to 70% and enables proportional schedulers to achieve their contracts. Anticipatory scheduling has been ported to Linux, where it is now the default disk scheduler.

Contents

Figures

Chapter 1

Introduction

Since their inception, operating systems have been designed to virtualize hardware and mediate resource access, and thereby enable and try to be fair towards multiple applications running concurrently on a computer system. A slew of techniques like CPU and disk scheduling, virtual memory, network file systems, and device drivers have been invented to enable operating systems to mimic standalone runtime environments for applications.

Despite the existence of these mechanisms, there is fundamental tension between software engineering considerations on one hand, and performance and fairness concerns on the other. Software engineering requirements of abstraction, modularity, information hiding between the operating system and applications, narrow system interfaces, etc. often render the application oblivious of certain critical aspects of its runtime environment. As a result, one application may need some resource more urgently than another application does, but ignorance of this resource contention can cause resource misallocation and dramatic performance anomalies.

For example, an application may hold on to a large amount of memory but be unaware that another application is in need of memory, or that the system is thrashing. Another example is in the disk scheduling world, where the operating system can be

unaware of applications issuing streams of synchronous I/O, and could consistently misallocate the disk resource to applications other than the one that issued the last request; this can result in very poor performance due to numerous unnecessary head seeks.

This dissertation analyzes two such kinds of resource misallocation arising due to the manner in which modern general-purpose operating systems enable main memory and disk access to be shared between applications. We name these problems *rigidity in memory management* and *deceptive idleness in the disk scheduler* respectively. We propose two systems, viz. *adaptive memory management* and *anticipatory disk scheduling* that solve these two problems. These systems exploit the possibility of closer coordination between the operating system and applications and between applications, to alleviate the problem between them. They improve the performance, fairness, and robustness of the operating system for a wide range of workloads without compromising on software engineering concerns.

1.1 Adaptive Memory Management

Many software applications can trade physical memory consumption for other resources. For instance, garbage collected language runtimes can trade collection overhead for heap size, and many programs can improve their performance by caching data that was precomputed, read from disk, or received from the network.

Unfortunately, operating systems do not provide to applications quantitative in-

formation about the cost of using memory. Free memory falling short is not a good signal for when applications should release their memory, for two reasons: (a) modern systems rarely have much memory that is actually free; rather, their memory often contains reusable data, and it makes more sense to talk of the value of a piece of memory, and how it relates to the potential cost of using the memory for some other purpose, and (b) when applications run concurrently, the amount of free memory in the system is not a useful indicator for how memory should be allocated between the two; again, a metric that would be useful to decide this would be the cost of using memory, which in some sense characterizes the degree of severity of memory pressure.

Due to lack of coordination between the kernel and applications, the latter become ignorant of the severity of memory pressure. Consequently, applications are forced to be conservative about their memory usage, including those that can allocate more memory when available to improve performance, or release it when the system suffers from shortage. Many memory intensive applications like databases and language runtimes need tedious experimentation to configure their memory footprints. Applications like Adobe Photoshop manage their own memory so as to be relatively independent of the operating system (especially primitive ones); however, they receive no information from the operating system about how much memory they should manage, or in what fashion. Despite being conservative, a load burst can force inflexible applications to exhaust system memory and incur paging. Finally, current operating systems lack mechanisms to control memory allocation in a manner similar

to 'nice' for CPU. Our proposed solution, viz. *adaptive memory management*, is a technique that informs interested applications about the degree of contention in physical memory using a severity metric, and allows applications and system libraries to react to it. Thus, applications can extravagantly allocate memory and release some of it under pressure, to reach a system-wide equilibrium that balances the impact of memory pressure on each application.

Our system encompasses four essential components: an appropriately chosen severity metric for memory pressure that is disseminated to all applications, a scheme for applications to react to it, a method for prioritizing applications in terms of memory consumption, and a kernel-level page replacement strategy that responds to the severity metric and enforces application priorities.

Our prototype implementation allows applications to extravagantly allocate memory when possible, and produces performance improvements in memory intensive workloads like databases (20% speedups compared to a conservative cache size) and java virtual machines (factor of 2 to 4 speedups due to far fewer garbage collections). It reduces the need for memory related configuration, improves system stability and responsiveness due to paging avoidance, and enables finer control on application performance by scaling the metric before presenting it to applications.

1.2 Anticipatory Disk Scheduling

Disk schedulers in current operating systems are generally work-conserving, i.e., they schedule a request as soon as the previous request has finished. Such schedulers often require multiple outstanding requests from each process to meet system-level goals of performance and quality of service. Unfortunately, many common applications issue disk read requests in a synchronous manner, interspersing successive requests with short periods of computation. The scheduler chooses the next request too early; this induces 'deceptive idleness', a condition where the scheduler incorrectly assumes that the last request issuing process has no further requests, and becomes forced to switch to a request from another process. We propose the 'anticipatory disk scheduling framework' to transparently solve this problem based on the non-work-conserving scheduling discipline. We sometimes introduce a small, controlled delay into the disk scheduler before it makes a scheduling decision, if we believe that the current request is likely to be followed by another from the same process within a short period of time.

Our prototype implementation produces performance benefits of up to 70% on disk intensive workloads like databases, webservers, and fileservers, and enables proportional-share schedulers to achieve their contracts efficiently. Anticipatory scheduling has since been ported to Linux, where it is now the default disk scheduler.

1.3 Thesis Contributions

The contributions of this dissertation are twofold:

- Identification of a problem that we call *rigidity* of physical memory management in general-purpose operating systems, its solution through the proposed *adaptive memory management* technique, and a prototype implementation for the FreeBSD operating system.

- Identification of a problem that we call *deceptive idleness* in the disk schedulers of general-purpose operating systems, and its solution through a new technique called *anticipatory disk scheduling*. An implementation of this technique has since been independently deployed in the Linux kernel, and has received considerable positive feedback from the user community.

Early work on the anticipatory scheduling technique was presented in the author's Masters thesis [51], and is summarized here for completeness. This dissertation extends that work by pulling in three years of experience of the Linux community with an independently deployed Linux implementation anticipatory scheduling (which has since been the default Linux disk scheduler). We will (a) describe differences between our prototype and this Linux implementation, (b) explore a new and important condition of disk write-intensive workloads starving read-intensive and real-time, which anticipatory scheduling solves (this is what first caught the Linux community's attention), and (c) understand the remarkable performance benefits reported in the Linux

community.

1.4 Thesis Structure

Chapter 2 provides background about the two problems useful for understanding the rest of this thesis. The chapters following that describe the two problems with their respective solutions.

Chapter 3 describes and motivates the problem in physical memory management that we term *rigidity*, and outlines our core ideas for an adaptive solution. Chapter 4 proposes the *severity metric* that is the key enabler for this form of adaptation, and Chapter 5 uses this metric to construct a page replacement framework. Chapter 6 presents results of an experimental evaluation of this approach, and demonstrates how we increase application performance in the common case through extravagant use of memory, adapt to reduce the need for footprint configuration, and improve robustness in low memory situations.

Chapter 7 describes the problem that we name *deceptive idleness* in the disk scheduling subsystem. Chapter 8 describes *anticipatory disk scheduling* as a solution, and Chapter 9 presents an experimental evaluation for it.

Chapter 11 describes the state-of-the-art in both these areas, and Chapter 12 concludes this dissertation.

Chapter 2

Background

This chapter explains some concepts in physical memory management and disk scheduling, and some terminology we will use throughout this thesis document. This chapter is complemented by a detailed description of related work in Chapter 11 and historical notes in some other chapters relevant to the discussion at hand.

2.1 Basics of physical memory management

Physical memory has become incredibly cheap over the decades, reaching prices as low as $40 for 1GB of RAM [86]. Most modern applications are content with configuring themselves with a reasonably large footprint that does not risk the possibility of overflowing memory. The dark side is, however, that many performance-minded applications achieve peak performance by using as much memory as possible, and there is abundant literature about the complexity of manually configuring footprints for database and virtual machines which, in an ideal world, should be completely solved between the operating system and the application [111, 14, 12]. Finally, even by applications being conservative about memory usage, a load burst can cause the collective working set of all applications to be larger than physical memory, and the system can thrash – this leads to the view that the memory problem is not completely solved, but

merely worked around using various compromises. This dissertation revisits memory management and proposes a solution that encompasses these problems.

2.1.1 Memory management in general-purpose operating systems

Memory management [109, 40] in general-purpose operating systems such as FreeBSD [3], 4.4 BSD [63], Linux [74, 42, 58], Solaris [62], Multics [30, 69], and VAX [56] consists of virtual and physical memory management. The former encompasses address translation, shared memory management, demand paging, etc. The latter can be decomposed into the following elements of core functionality:

Page allocation and deallocation: The operating system manages free space in units of physical memory pages, which are usually all of the same size unless Superpages are used [66]. When a process requests memory (either explicitly or by accessing virtual memory locations that are memory-mapped from files), physical pages are obtained from the free pool and mapped into the process address space in batches, and the virtual address of these pages is provided to the application. When a process releases memory, the corresponding physical pages are either immediately or lazily returned to the system's free pool [109].

Page preemption: If the free memory pool becomes empty, subsequent memory allocation requests are satisfied by preempting some memory already allocated to applications. The next section describes the basic design choices that page replace-

ment systems make, but they share the common trait of identifying pages that are good candidates for replacement, clearing their contents through some mechanism, and returning them to the free pool. If the page selected to be replaced has contents identical to a copy on disk, then it is immediately freed and the application experiences a page fault the next time it accesses the location. If the page is modified, then its contents are written out to swap space, and then the page is freed.

Pages are typically chosen for replacement based on some variant of LRU, using the assumption that pages that were not recently accessed will not be accessed again in the near future. Remembering the accurate sequence of page accesses is very expensive, so most operating systems implement an approximation like the Clock algorithm, based on sampling reference information to the pages [109]. On many architectures this is possible through page-level reference bits that are set upon accesses to the page, whereas on architectures like the Alpha's, a trap is raised upon page accesses so that the handler can mark the page as referenced [100].

Under memory pressure, the Clock algorithm scans pages to identify pages without their referenced bit set, or clears the bit of those who have it set to give them a *second chance* to remain in memory. Pages that are so identified to have not been recently accessed are first cleaned if modified, and then freed and reused [109]. In the page replacement scheme that we propose in Section 5.1.1, we employ a similar scheme for scanning reference bits to monitor the approximate time since last access to pages, but we scan them at a rate proportional to what we believe their time since last

access was. Furthermore, we maintain pages in per-process bins so that, rather than selecting non-recently-referenced pages for replacement (which are an approximation to the oldest page), we are able to variously scale the page ages for each process to select the page with greatest weighted age.

2.1.2 Taxonomy of memory management systems

The primary objective of physical memory management systems is to efficiently apportion memory between applications in a manner that is fair between the participating applications, conducive to high performance for these applications, and robust during system-wide memory shortages. The fairness objective includes the consideration that a malicious application should not take a disproportionate amount of memory away from others.

Global vs. process-local page replacement: One key distinction between different types of memory management systems is the choice of whether they perform page replacement at a global or process-local granularity. The latter model is adopted by operating systems such as Nemesis [44], VAX [56], and others [113] with the express objective of isolating processes from each other. The second benefit of these schemes is some degree of user control on memory allocations for each process. However, from an overall performance standpoint, when applications responsibly use memory, global page replacement is preferable because it has the ability to select the least valuable page across the entire system for replacement.

Application-assisted vs. transparent page replacement: A second dimension for characterizing memory managers is whether or not they interact with applications regarding the choice of pages to replace. Most popular operating systems choose not to have their page replacement schemes interact with applications at all; this is usually preferable to applications from the software engineering perspective of having a narrow interface to the kernel.

On the other hand, various page replacement systems, mostly developed in the research community, are implemented (to different degrees) within application code. Examples of this are the Mach OS [102], the SPIN OS [16], the Exokernel OS [35], application-controlled file caching [24], and external page cache management [45].

The system we propose in this thesis takes a radically different stand. We have the system notify applications of the level of memory pressure. Applications get the option of responding to these notifications by freeing some memory depending on which and how much memory they believe is cheaper to release than to suffer the consequences of memory pressure to the degree that it is present. Thus, the interface between the application and the kernel is kept relatively narrow. This facilitates programs written in high-level languages like Java or perl to receive notifications of memory pressure, and enables them to free some buffers that they believe to be less expensive. Note that it is much easier for an application to respond to notifications of the level of memory pressure than having to deal with physical or virtual page addresses or swap locations.

Thrashing avoidance and control mechanisms: Some page replacement systems incorporate mechanisms to detect, avoid, and sometimes control thrashing. Multics [30] identifies *working sets* of each process, to determine a set of processes that can fit in memory [32, 34, 33], and dynamically partition physical memory among them. At runtime, if applications start accessing more memory and touch larger working sets, then the operating system tries to determine if there is sufficient memory in the system to accommodate these applications. If the operating system decides otherwise, it may employ a technique such as *swapping* to give each application some time to operate from main memory, while it keeps some other applications dormant and swapped out [30, 62]. Finally, WSClock is an algorithm that was proposed to combine working sets with Clock, in an attempt to perform global page replacement with some degree of thrashing avoidance [25].

These thrashing avoidance methods have limited relevance for two major reasons:

- Systems now have three orders more memory than they did just a decade ago, and correspondingly, the risk of overflowing it has dropped. This has resulted in a dichotomy of concerns – modern, especially interactive applications generally do not need to worry about memory overflow. At the same time, distributed and server-side applications do have to be absolutely sure that their activities do not result in thrashing. Such applications find limited use in a scheme to alleviate the impact of thrashing; instead, they would find a feedback mechanism for memory pressure more useful in addressing their concern.

- These thrashing avoidance schemes were proposed in the era of multiprogrammed batch workloads, whose access patterns were static and predictable compared to modern applications for which it is difficult to predict the working set.

Thrashing avoidance, and more importantly, to remove applications' concern about the risk of thrashing, is one of the goals of the system proposed in this thesis. However, our approach is not to swap out processes that cause (or might cause) the system to thrash, but to encourage them to release some memory *before* the system comes close to the point of thrashing, as qualified by our severity metric for memory pressure.

2.2 Basics of disk scheduling

The performance gap between disks and other components of a computer system is large, and has been steadily widening over the years. Disk speeds are growing by only about 7% a year, and are not keeping pace with the 55% annual increase in CPU speeds. A substantial fraction of modern day systems are disk intensive, such as database systems, fileservers, large working-set web and ftp servers, and various types of scientific computation. New application types like multimedia clients and servers are driving the need for differentiated quality of service in the disk subsystem. It is thus becoming increasingly important for modern operating systems to provide high performance and predictable quality of service to applications.

The disk subsystem in an operating system can be viewed as a means of delivering

disk requests from applications to the disk controller. A *disk scheduler* is a vital component of the disk subsystem, whose goal is to reorder these requests for reasons of achieving higher performance, satisfying quality of service requirements, etc. This chapter presents an overview of disk schedulers as implemented in most operating systems.

Consider a single physical disk attached to a computer system. Applications invoke system calls like `read` and `write` to gain access to files. These are sometimes serviced from the filesystem cache that the kernel maintains; misses in this cache become *disk requests*. Applications may similarly map files into memory, and thereby access this memory region. This causes page faults that may sometimes also result in disk I/O. Most operating systems perform background maintenance tasks such as swap-space management and disk defragmentation. These can cause disk I/O that is not directly initiated by applications.

These requests are serviced by the disk one after another. This sequence is determined by the disk scheduling policy, and thus, by the disk scheduler implementation. The *disk scheduler* is thus an artifact of the operating system that receives multiple requests, queues them internally in some fashion, and dispatches them one by one to the disk driver for (usually) immediate service.

The disk scheduling paradigm differs from the related problem of CPU scheduling in the following fundamental ways. Disk requests are explicitly generated by applications. These requests are associated with two metrics: a request size, which is known

in advance, and a request service time. These respectively result in performance measures like *throughput* in MB/s and *disk utilization* in fraction of disk time spent servicing requests*. Disk scheduling is a *non-preemptive discipline*, i.e. a request, once dispatched to the disk for service, cannot be withdrawn. Context switching overheads for disk take the form of seek and rotational latencies, and may range from very small to very large, depending on request placement. In contrast, CPU scheduling is non-request-driven, but is based on time being allocated to processes. This is a preemptible scheduling discipline: the processor can be relinquished early or the quantum can be forcibly preempted. Context switching in CPU is moderately expensive, but does not possess as much potential for variation as switching between disk requests. Such differences lead to problems unique to disk scheduling, such as deceptive idleness.

A disk scheduler is structurally composed of two pieces: the *disk scheduling framework*, and an implementation of the *scheduling policy*. The framework provides the required interface between the scheduling policy and the rest of the kernel. The policy defines an ordering strategy for disk requests, and performs this reordering when the scheduling framework invokes various *scheduler-specific methods*. Replacing one scheduling policy by another can be achieved by keeping the framework constant and replacing the scheduler-specific methods.

A scheduler is said to be *work-conserving* if it never leaves the disk idle when

*not the fraction of time spent doing *useful* work.

there are requests waiting to receive service. A *non-work-conserving* scheduler may allow the disk to become idle, despite the presence of pending requests. The next section examines the structure and functioning of a traditional work-conserving disk scheduling framework.

2.2.1 The scheduling framework

The scheduling framework provides the infrastructure that receives requests from upper layers of the kernel, dispatches requests to the disk driver, handles completion of request service, etc.

Figure 2.1 : State transitions for a conventional scheduler

The disk scheduler operates in one of two states: IDLE and BUSY, which reflect the state of the physical disk at that instant (figure 2.1). The disk is initially idle. Requests arrive at the scheduler at any time; these are enqueued into the request pool using the scheduler-specific `sched_enqueue(req)` method. If the disk was idle at this moment, the work-conserving scheduler is compelled to issue a request to the disk immediately, thus calling `schedule`. This moment in time is a *decision point*. The scheduler chooses a request using the `sched_choose()` method, removes the request from the pool using `sched_dequeue(req)`, switches state to BUSY, and dispatches the

request to the disk driver for immediate service[†]. When a request finishes receiving service, the scheduler state is switched back to IDLE and the `sched_finish(req)` method is invoked if the scheduling policy wishes to be notified of this event. Then `schedule` is called again if any more requests are found pending (figure 2.2).

```
enum STATE { IDLE, BUSY } state = IDLE;
integer pending = 0;

issue(new): // invoked by upper layers of the kernel
    sched_enqueue(new);
    pending++;
    if (state != BUSY) schedule();

schedule(): // local function
    assert(pending > 0);
    assert(state == IDLE);
    next = sched_choose();
    sched_dequeue(next);
    pending--;
    state = BUSY;
    // send to disk driver for immediate service
    PERFORM_DISK_IO(next);

finish(req): // called from the disk interrupt handler
    assert(state == BUSY);
    state = IDLE;
    sched_finish(req); // if the scheduler wants to know
    if (pending) schedule();
```

Figure 2.2 : A traditional work-conserving disk scheduler

2.2.2 Seek-reducing scheduling policies

The simplest disk scheduling policy is First Come First Served (FCFS), wherein `sched_enqueue` and `sched_dequeue` just maintain a FIFO queue. However, this policy

[†]SCSI disks implement tagged queueing, wherein multiple requests are scheduled to the controller, internally seek optimized and then serviced. We discuss this later.

is efficient only for very light workloads, with little concurrency between requests. Magnetic disks with movable heads traditionally incur large overheads for positioning the disk head over the desired sector. In order to amortize this cost over larger data transfers, disk schedulers implement different kinds of seek reduction policies.

To give an example, a 7200 rpm IBM Deskstar 34GXP disk sustains a continuous 21 MB/s of sequential data transfer, but only about 5 MB/s of random-access 64k block transfer. This is because a 64k chunk takes 3ms to read from this disk, as compared to an average seek time of about 9ms.

The underlying idea in all seek-reducing schedulers is thus to capitalize on spatial locality of disk access. If a request issuer generates requests that are targetted at nearby disk blocks, then the scheduler should be able to service multiple requests without much physical movement between requests. This means that the head should not need to seek much from one track to another, and the disk platter should not need to rotate much to reach the right sector. Schedulers take a variety of approaches to implement this desired behavior.

The Shortest Positioning-Time First (SPTF) policy greedily schedules the available request with the smallest head repositioning time. To eliminate the possibility of starving distant requests, Aged-SPTF (ASPTF) assigns increasing priority to requests that have been kept pending for long. A scheduling policy commonly implemented in UNIX-based systems is the Elevator algorithm (C-LOOK), which orders disk requests such that they would receive service them in strictly monotonic (say increasing)

order of their disk coordinate. When the last request in this sequence receives service, C-LOOK dispatches the request at the smallest disk coordinate, thereby forcing the head to perform potentially a long seek back before starting over [119, 52, 96]. Chapter 11 discusses these in greater detail.

2.2.3 Proportional-share scheduling policies

The evolution of the Internet brought with it a slew of interactive and supporting server-side applications that demanded predictable performance from the operating system. This need for differentiated quality of service support in operating systems has as one of its requirements appropriate management of all system resources, including disk service. Disk schedulers are thus becoming required to provide more functionality than just raw throughput, namely low, predictable, and controllable response times.

The central theme in differentiated QoS is to provide support for *multiple service classes*, and allow applications associated with each service class to be treated according to some *service agreement*. Traditional operating systems multiplex system resources among resource principals such as processes and threads. Banga et al. [10] proposed the *resource container* abstraction that separates the notion of a resource principal from a process or a thread. We use resource containers as our service class abstraction.

Among various types of differentiated service, this thesis mainly examines the im-

pact of deceptive idleness on proportional-share scheduling. The service agreement in proportional-share disk schedulers is a partition of total disk service to the active service classes in some required proportion, expected to be satisfied over some pre-specified time-scale. Several proportional-share scheduling algorithms have been proposed in the literature; some of these are described in chapter 11.

This thesis uses the Stride scheduler [117, 115], trivially adapted for disk scheduling. This algorithm performs deterministic proportional-time scheduling, i.e. it proportionally distributes disk service time (and thus *disk utilization*) among resource containers on a fine granularity. It maintains a separate virtual clock per resource container, and increments it on completion of a request attached to that container. The increment is proportional to the request service time, and is inversely related to the share assigned to the container. At decision points, a request is selected from the container with the smallest virtual clock. A container remaining idle for an extended period of time explicitly leaves; upon returning, it is assigned a virtual clock equal to a global clock that corresponds to the smallest virtual clock of all active containers.

A direct adaptation of Stride from CPU to disk scheduling would try to achieve accurate proportions but not exploit any opportunities for seek optimization. To avoid this performance penalty, some of our experiments implement a *relaxed* variant of the above, similar to Yet-another Fair Queueing (YFQ) [20] and the PIso disk scheduler [113], which simultaneously perform seek reduction and proportional scheduling. At decision points, these schedulers pick a request within a threshold of the minimum

virtual clock of all resource classes, which has the minimum positioning time from the current head position.

2.3 Terminology

Applications are referred to in this thesis as first-class entities that consume resources of some type, e.g., memory or disk. Therefore, depending on its structure, an application could map to a process or a thread or a resource container or whatever fits the context.

Elastic applications are those with the ability to control the usage of some resource, such as memory, at the cost of performance or the usage of some other resource.

Physical memory manager is the operating system component that handles allocation, release, and preemption (page replacement) of physical memory.

Page daemon is the operating system component that selects application pages to replace, and evicts them as allowed by the characteristics of the page. If the page has been modified in memory, it issues a disk I/O to page it out to a backing file if available, otherwise to swap space.

Memory pressure is a condition where applications' collective need for physical memory exceeds its availability.

Page age is the time since last access of a physical memory page. We say a page is *old* if it has a large page age.

Severity of memory pressure is a metric that we devise to quantify memory pressure in terms of the potential amortized cost of paging out an equivalent amount of memory. It is expressed in time units. A smaller value indicates higher memory pressure, because the oldest page in the system would be relatively recently accessed.

Nice value is the UNIX notation for CPU scheduling priorities, and is expressed as an integer value between -20 (highest) to +20 (lowest).

Disk requests are requests for disk I/O issued to the disk subsystem. A disk request is guaranteed to reach the disk hardware and not be serviced from memory. These requests can initiate read or write I/O for an arbitrary number of bytes, up to a predefined maximum transfer size.

A disk scheduler is an operating system artifact that schedules disk requests for service on the available disk(s). For the purposes of this thesis, we restrict ourselves to a single physical disk.

A resource principal for a disk request is an entity that is ascribed the responsibility of having initiated the request. This could be the process, thread, resource container, page daemon, VM subsystem, etc. as is appropriate for the specific type of request and its issuer.

Idle resource principal: A resource principal is *idle* with respect to disk activity at some time instant if it did not issue any disk requests despite having had the opportunity to do so since the time its last disk request got serviced (i.e. it got CPU scheduled, did not block on network I/O or user input or any other possible bottleneck).

In particular, having no outstanding requests pending at a certain moment in time is not sufficient evidence of the resource principal being idle, because it may not have received a CPU quantum since its last disk request.

Chapter 3

Rigidity versus adaptivity in memory management

This chapter describes and motivates the problem of rigidity in memory management, and outlines the key ideas towards our adaptive solution.

3.1 Problem description

In the early days of computing, the limited amount of physical memory posed a hard limit on the size of programs. Computer programmers of the 50s started using secondary storage as an extension to physical memory (magnetic drums since 1951, and magnetic tapes since 1957 [23]).

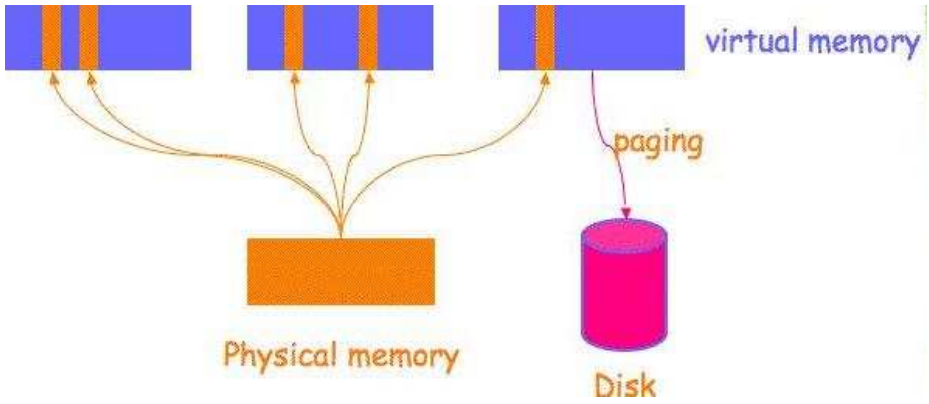


Figure 3.1 : Virtual Memory

Virtual memory was invented in 1961 for the Atlas computer [37], as a means of

presenting applications with a large virtual address space. Pages in virtual memory may be either mapped to physical memory pages, or transparently paged out to secondary storage (see Figure 3.1). This paradigm shift allowed the complexity of memory hierarchy management to be migrated from applications to the operating system, and thus vastly simplified programming in the 60s and thereafter.

However, it became clear that beyond the limited improvements obtained by optimizing the page replacement policy, there was limited scope for bridging the huge performance gap between secondary storage and the fast-but-small main memory. Essentially, the virtualization of physical memory was only for convenience, and was not effective from a performance standpoint. Therefore, focus shifted to the policy question of automatically deciding which pages to retain in main memory and which ones to dispatch to swap. This led to debate in the 60s and 70s, resulting in numerous page replacement policies such as Working sets [32, 34, 33] that identified performance-critical pages to be kept in memory. Techniques were also devised for gracefully handling low-memory situations, such as process swapping and swap rotation [30, 62].

Over the decades, increasing availability and decreasing cost of physical memory caused the chance of overflowing it to significantly decrease. However, people's expectations of computer systems also increased to a great extent over the mindset of the 70s. Therefore, the small but non-zero chance of paging if applications overflow physical memory came to be (rightfully) regarded as a serious, sometimes fatal

performance penalty. This has led application programmers to become cautious.

Applications are developed and subsequently run in environments often unforeseen at development time, and are subjected to unpredictable workloads. This compels programmers to be conservative about the memory usage of their applications, although they could often improve their performance by using more memory when it is available, and releasing it when there is memory pressure. A computer user or a system administrator may choose to tolerate some paging activity in exchange for a larger memory footprint, but a programmer has no foreknowledge of the stress on the systems on which their applications will be run, and is therefore forced to ensure that the application fits in memory. The fully transparent abstraction of virtual memory that was a boon for programming convenience is thus becoming a bane for the performance-minded programmer.

On the other hand, despite applications being conservative about memory usage, current systems do face the risk of incurring a load burst that may result in severe paging, which can degrade performance to unacceptable levels. Some of this paging may be for application memory that could easily have been freed by the application with lower performance penalty than paging it out to disk and later paging it back in. Therefore, we contend that the virtual memory abstraction renders applications oblivious of the contention for physical memory. This especially includes those applications that can use some simple memory pressure information to balance memory pressure within themselves against memory pressure within the system, in order to

improve their performance and the system's robustness.

3.2 Solution insight: Leveraging elastic applications

This thesis proposes a system called *adaptive memory management*. The central idea is to provide feedback to applications in the form of a *severity metric*. This metric is designed to quantify the current cost of reclaiming memory, in a manner that gives applications the basis for computing the tradeoffs in adapting their memory consumption. In normal operation without memory pressure, the system encourages applications to step up their memory usage and benefit by it. When there is memory pressure the system expects them to release some memory, and tries to achieve an equilibrium state in which applications perform *global least-impact-first memory replacement* to balance their internal memory pressure against the global pressure within the system. In this manner, we try to harness the following *elasticity* property inherent to many modern applications.

Many modern applications (and their libraries and runtime environments) use algorithms that are *elastic* in terms of their ability to trade memory consumption for performance or the usage of some other resource(s), like CPU, disk, or network bandwidth. Here are some examples that describe common types of applications that can achieve performance gains using a substantially large memory footprint, but would prefer to release most of this memory than to have it paged out.

Language runtimes such as Java virtual machines perform garbage collection with

considerably less overhead if they use a large heap; in times of memory pressure they can quickly release this unused memory. Many malloc library implementations tend to hold on to memory that applications free, so as to minimize their interaction with the system; some malloc implementations can release this memory to the system if desired. Databases, web caches and web browsers that maintain memory caches of disk and network data can increase their performance (often substantially) by enlarging their caches, or alleviate memory pressure by releasing a fraction of the cache that is cheaper to regenerate than page out. Moreover, databases, Java runtimes etc. can configure their footprints without need for manual configuration. Applications like Adobe Photoshop stand to gain by caching a large amount of intermediate data, but can free them if the system is running out of memory. Many scientific applications can trade CPU time for memory by retaining different amounts of partial results that may be reused at later stages in the computation. They often also have the ability to vary their problem size, or in the case of simulation applications, some of its parameters (such as accuracy) in reaction to memory pressure.

While it only takes small changes in applications and/or shared libraries (i.e. transparent to applications) to support these memory-related adaptations, it is unfortunately nontrivial to decide when to invoke these adaptations, especially ones that have significant impact on the application's performance. This is because applications have no means of comparing against other applications on the system, to gauge the relative worth of their memory.

In our solution to this problem, we develop and evaluate a method that enables us to take advantage of such elasticity in common applications, with the high-level goals of improving application performance, system robustness under memory pressure, and user control over memory allocations.

3.3 Analogy: Water levels

The concept of balancing memory pressure and the nature of the severity metric can be illustrated using the following analogy.

Imagine cylindrical containers of different cross-section areas, standing alongside each other, with water at different levels (Figure 3.2). Connect these containers using a narrow tube. Water flows from the container with a higher water level to one with a lower one, until the overall level equalizes.

Gravity is the reason this equalization happens, but the net outcome is that (a) the overall water pressure minimizes, and (b) the base of every cylinder reaches the same water pressure. This was possible due to collaboration among containers, and the communication of a sense of a *global water level* to them so that they had the basis to collaborate. In particular, each container did not need to be informed about the existence of, or the water level of every other container. Further, note that the metric needed to minimize pressure is the *level* of water, not its volume.

We will revisit this analogy in Section 4.1.1 as we develop the precise manifestation of the memory pressure severity metric.

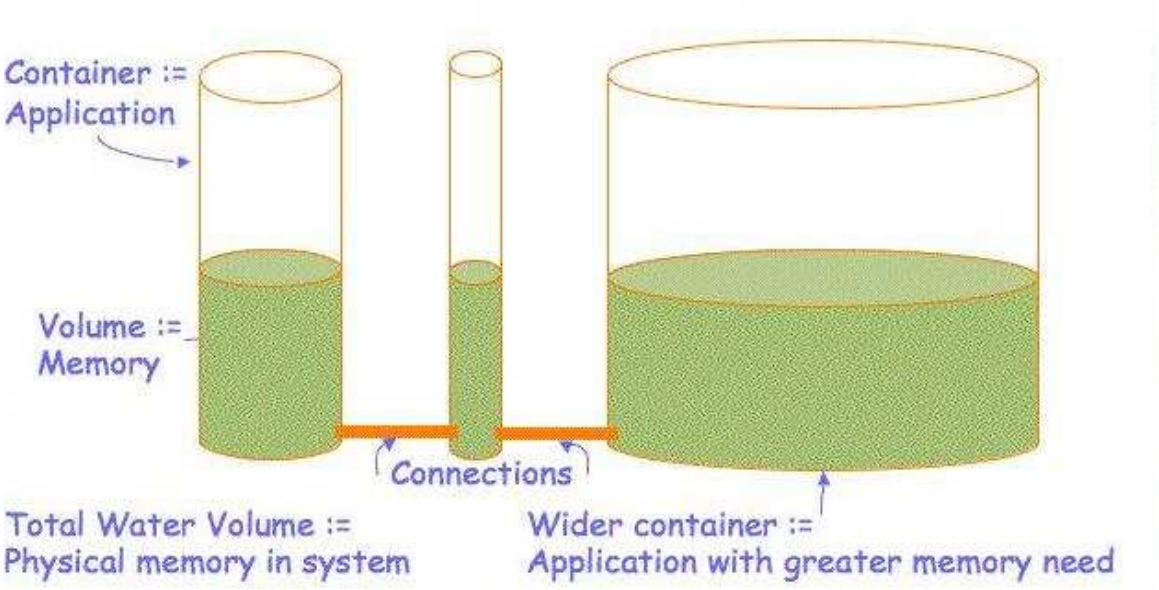


Figure 3.2 : Water levels reaching a natural equlibrium

3.3.1 Water levels under different gravities

Taking this analogy further, if it were possible to individually scale up or down the gravitational force experienced by each container, then water levels would reach a new equlibrium (see Figure 3.3). This would have different levels in each container, but the pressure at the bottom of each container would nonetheless be equal. Thus, equalizing pressure at the bottom rather than level can be seen as the objective that both these situations converge to.

This is the state of (weighted) equlibrium that we hope to achieve with applications and memory allocations – that each application uses as much memory as what results in the impact of memory pressure on them being equivalent.

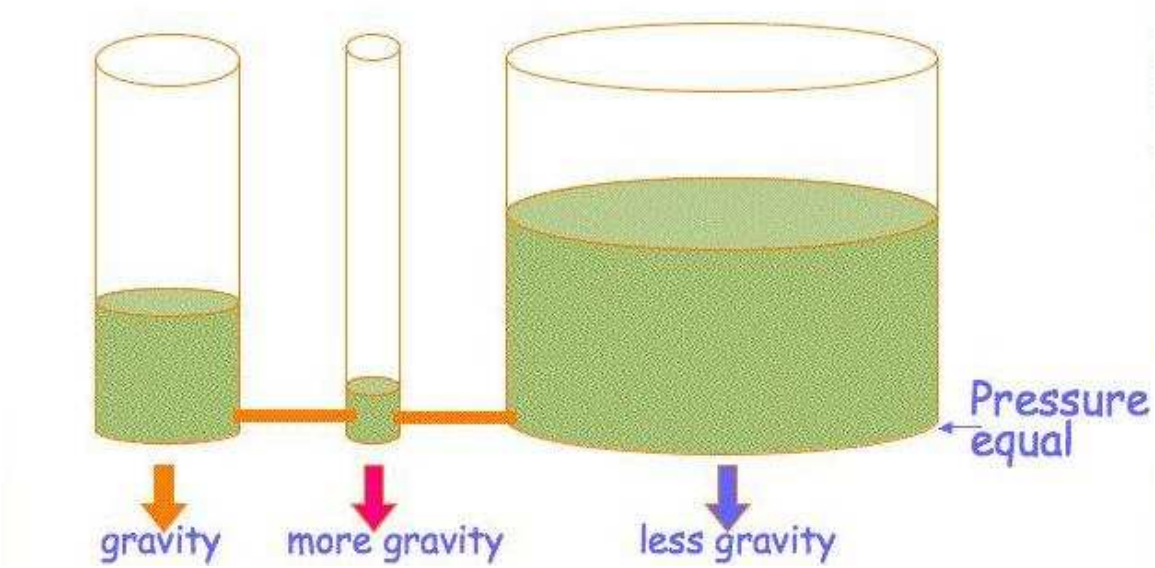


Figure 3.3 : Water levels with different gravity exerted per container.

3.3.2 Incentives

Continuing with this analogy, on being informed of the global water level, if a narrow container with a tall water column cannot give up water (say because its vent is clogged), then that is only a minor loss for the system as a whole, because there was only a small amount of water in that narrow container.

On the other hand, if a wide container cannot give up water, then that's a big hit on the system. However, if there is no way to coax it to give up any water because it genuinely needs to hold all that water (for some reason), then in some sense there is nothing to be done, and the system would have to live with the pressure. Taking a broader view, if a hypothetical operator of this water-container system feels that the wide container is unfairly holding too much water compared to its legitimate

need, then he/she may feel justified in *forcing* the container to release water, say by taking a cup and pouring water out of the wide container, or more drastically, by breaking the wide container to distribute its water among others (see Figure 3.4. Avoiding such drastic penalty can be construed as incentive for the wide container to cooperate and release water when hooked up to other containers. This scheme is based on incentives; it intentionally does not mandate every container to release as much water as requested, because it accommodates the possibility that a container may have a disproportionately large but legitimate need for retaining water.

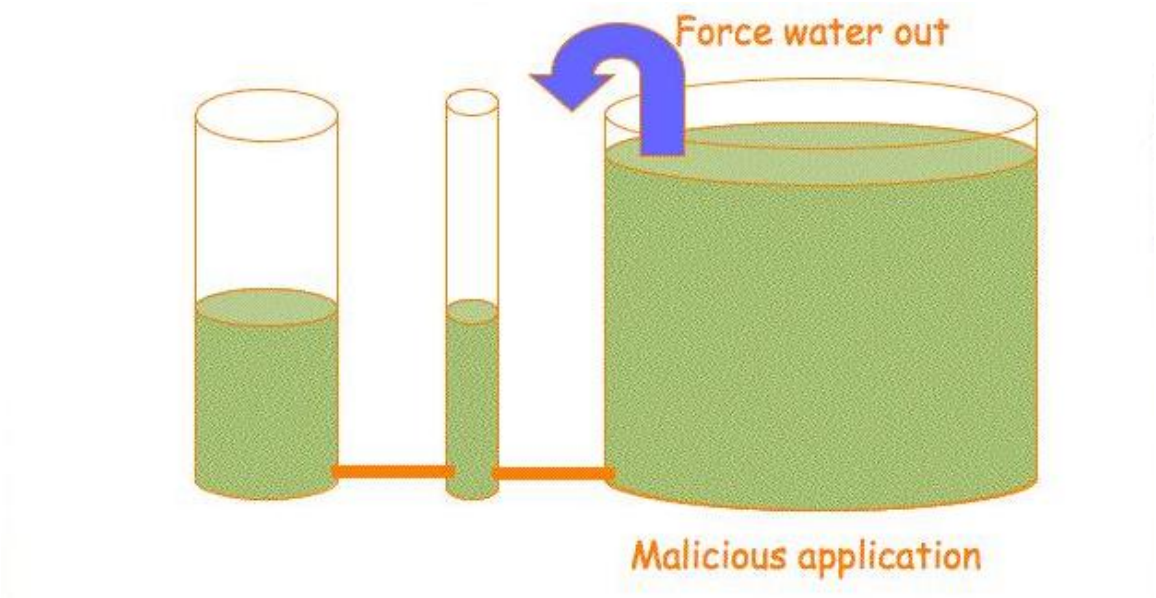


Figure 3.4 : Dealing with non-cooperative applications.

Translating this to the memory management world, if some application is inflexible with respect to memory, then the system will live with that – grudgingly if the application consumes a lot of memory. However, if some application selfishly or

maliciously holds on to a lot of memory, then given suitable mechanisms to force memory out of such an application, it becomes the user's (operator's) decision to do so – and indirectly, this becomes the application's incentive to cooperate. These are some of the essential ideas that we will develop throughout this chapter of the dissertation. Section 4.1.1 draws explicit parallels from concepts like water level, pressure and volume to the memory management world.

3.4 Key ideas

Our scheme of system support for memory adaptation is based on the following four ideas:

(1) Notifications: The system can *notify* applications about impending memory pressure, and give applications a chance to react to it. Being aware of this, elastic applications can safely allocate large amounts of memory; when notified, they should release memory by performing actions that are less costly than paging. These applications may restore their memory allocations if free memory becomes available again, thus operating the system close to full memory utilization.

(2) Severity metric: Furthermore, the system can expose information about the *severity of memory pressure*, using an appropriately chosen metric. This enables applications to compare the cost of performing a certain memory releasing action normalized against the baseline of I/O costs being incurred (or potentially incurred)

for paging, so that they can gracefully respond to increasing memory pressure by taking increasingly drastic actions. As a result, this metric enables elastic applications to *independently* decide when and to what extent to perform their adaptations.

These two ideas – notifications along with severity information – can increase the performance of elastic applications when excess memory is available, as well as improve robustness in memory constrained circumstances.

(3) Weighted adaptations: A system may be running several elastic applications, each of which may autonomously choose among several actions to reduce memory consumption when notified by the system. It is therefore desirable to impose a usable and intuitive scheme of *preferences*, ultimately under user control, to determine which applications should first take actions and to what degree. For example, it should be possible to make background applications more aggressive about shrinking their caches than interactive applications.

(4) Weighted page replacement: Finally, a natural and useful corollary of this preferences model is to bias the page replacement policy with these preferences. If and when paging happens, this would, for instance, permit memory from interactive or soft real-time applications to be paged out only when memory pressure is severe, whereas background processes can become attractive candidates for page eviction.

The key insight that connects these four concepts is a *severity metric* that quanti-

fies the availability of free memory and the current cost of reclaiming memory, which we shall define based on the recency of access to the page that is the next candidate for replacement. This severity metric enables us to establish a quantitative relationship between the memory contention in the system, the type and degree of applications' reactions, and the preferences model.

This approach is complementary to application-controlled caching [24, 45] and paging [102, 16, 35], which allow applications to control the page replacement policy and the mechanism for storing the contents of replaced pages. We treat paging as just one mechanism to reclaim physical memory; in fact, during normal operation, our system would not page at all, but use adaptations of elastic applications as the predominant means for memory management.

3.5 Motivation

Physical memory has become cheap and plentiful over the decades, reaching prices as low as $40 for 1GB of RAM as of 2005 [86]. There was intense debate in the 50's and 60's over memory management policies that avoid thrashing and minimize paging for scientific and business computing type workloads. Today, servers and even desktop machines are equipped with several GB of RAM, leading to the prevalent notion that memory shortages can be easily eliminated by throwing in more memory, and that systems that exhaust their memory are essentially misconfigured. We refute this view, and present the following key motivations for the problem that we address.

3.5.1 Automating memory footprint configuration

To the extent possible, memory settings should be automatically configured. In general, it is well-documented that server administrators find it tedious to manually adjust the footprints of server applications for high memory utilization without paging.

i. For example, database caches need to be periodically and manually tuned for performance critical reasons, but this is known to be tricky [111, 14]. There are extensive and tedious instructions for assigning large heap sizes to Java virtual machines to reduce the frequency and overhead of garbage collection without paging [12]. It would be a win to reduce the number and demands of the memory-related knobs in these systems.

If such applications are run standalone on a system, then one can assign a cache/heap size smaller than free memory available to applications – although conservatively, since free memory may decrease due to allocations by the kernel or system services. However, these problems are far aggravated if (1) such applications are co-located with other, possibly elastic, applications (which could be more instances of the same application), and/or (2) there are dynamic workload fluctuations. It then becomes painful and suboptimal to tune the memory settings and achieve the right balance. Our system is designed to allow diverse applications to independently configure their own footprints, and to dynamically adjust them in reaction to memory pressure, by normalizing their memory adaptation costs against a common reference point of potential paging cost.

ii. Automatic memory configuration can benefit distributed applications, but even a simple memory pressure notification goes a long way. Distributed applications often have the ability to alter their memory requirements by adjusting their concurrency and parallelism. Moreover, due to variance in memory requirements between tasks, it becomes tedious and sometimes extremely wasteful to provision for the maximum memory needed, whereas a simple memory pressure notification could allow huge requests to be handled differently, say by truncating them (at the cost of determinism).

iii. Large-scale simulations are often memory intensive, and can sometimes influence their footprints by controlling their problem size. Users often go through painstaking iterations to guess a large enough problem size that will make the simulation just fit into memory. Instead, the simulation can be made portable across systems with different amounts of RAM, if it is designed to grow its problem size until notified of impending memory pressure, and then freeze the problem size and continue the simulation.

iv. Some specialized applications automate such adjustments in practice using thresholds on free memory polled via `top` or `vmstat` [12]. If the application is running in isolation and without any file cache, then this simple approach is sufficient. However, if there are multiple applications competing for memory, then merely knowing the amount of free memory in the system is not enough for applications to achieve an effective balance (one that ensures least-impact-first replacement); instead, they should be informed of the cost of reclaiming memory, so that they can trade off this

cost against the benefit they obtain by holding the memory. For instance, memory that was accessed a long time ago may never be accessed again, and can be released or even paged out with low performance impact.

Secondly, there is typically not much memory that is actually free in a system that populates its file cache with many recently accessed files. Therefore, the `vmstat` approach of determining free memory does not allow applications to compete against the file cache to reclaim the cheapest memory buffers between them.

3.5.2 High memory utilization

One can always envision constructive uses for more memory. As mentioned earlier, elastic applications are currently forced to remain conservative about memory usage, and there are many practical instances (some depicted in Chapter 6) where using large amounts of memory can significantly benefit application performance. The operating system can also make use of extra memory by aggressively prefetching and caching file system blocks [99], especially if the system has cheap background I/O mechanisms like freeblock scheduling [59]. Log-structured file systems are based on the assumption that files are cached in main memory and that increasing memory sizes will make the caches more and more effective at satisfying read requests [91, 71].

In summary, we envision many classes of future systems to operate in a regime where main memory is deliberately utilized to a high degree, and benefit to varying

amounts in performance as a result of this usage. Aggressive caching and prefetching, infrequent garbage collection, and thrashing avoidance can be deployed to help system performance in many ways. All this is feasible only if applications are able to depend on a system facility that monitors memory pressure, and informs them about when to allocate or release memory – i.e. which "closes the memory allocation–usage loop".

3.5.3 Graceful adaptations

For memory adaptations to be the primary mode of memory management during normal system operation, they need to be graceful and continuous, and should frequently happen in numerous applications. The key to gradual adaptations is a continuous severity metric that quantifies the cost of reclaiming memory, in a manner that applications can independently and gracefully react to, such as by varying its cache size by small amounts. Contrast this with a naive scheme of memory adaptation that avoids thrashing by notifying all applications each time there is memory pressure, and they aggressively react by freeing all the memory they can. Such a system can also suffer from oscillations, with alternating regimes of memory allocation and release. Clearly, our graceful adaptation approach is preferable to this naive scheme if systems are to run at high memory usage, and are to support diverse adaptation methods with unequal performance impact.

3.5.4 Accommodating load bursts

Although computer systems usually have sufficient memory to avoid paging, it is al-

ways possible to encounter an unexpected load burst that can cause a disastrous performance degradation. Workstations, laptops and various slim devices may have gigabytes of RAM at the time of purchase, but this RAM often becomes insufficient after some years of software installations and upgrades. When systems become sluggish, users are naturally accustomed to making them more responsive by closing windows or suspending applications; we contend that elastic applications should be exploited to make such intervention largely unnecessary.

Large server farms are often economically provisioned with resources that just exceed their expected high-end requirements. If such a system experiences a load spike, then we desire that the elastic applications on these machines shrink to gracefully accommodate the burst, rather than start thrashing or avoid thrashing by drastic measures such as killing applications.

3.5.5 "Nice" for memory

It is useful to extend CPU nice values to memory management. There has been a recent surge of interesting background applications for desktop platforms such as peer-to-peer file sharing and web caching, and distributed computations like BOINC applications, SETI@home, Folding@home, ClimatePrediction.net, etc [4]. Shared computing platforms like Condor [19], and more recently, HPL's Planetary Computing [95] and the Intel Research seeded PlanetLab [79] support guest applications with unpredictable memory requirements. Beyond isolating processes through ex-

plicit allocation limits and self-paging [44, 113], both with limited applicability, there has been limited work on user control over memory allocation. Using our "nice" for memory scheme, desktop users can prefer that the working sets of interactive applications be retained in memory, while permitting background applications to use copious amounts of memory when foreground activity is less memory intensive.

3.5.6 Operating system primitives for thrashing avoidance

Sometimes, users need to run applications dangerously close to the available memory, perhaps expect some amount of paging to happen, and wish for more assistance from the system in controlling the extent of paging. Such scenarios are common in large-scale simulations, where users are familiar with having to monitor memory availability and paging I/O, and manually suspend and resume processes lest they thrash. Memory pressure notifications can be used to automate this, and trigger simple corrective actions like taking turns to voluntarily sleep for reasonable periods of time while letting other jobs proceed.

Chapter 4

Severity metric

This chapter develops a metric to quantify the severity of memory pressure in the system at any given instant. This metric should be defined in such a way that applications that know its value are able to objectively and individually ascertain the extent to which they ought to respond to memory pressure. Once we have such a metric, the operating system can compute and continuously report it to all applications, and/or notify them whenever the metric changes substantially. Elastic applications can respond to it to achieve the goals laid out in the previous chapter.

We now postulate the following expectations of the severity metric, and of applications that know and respond to it. Afterwards, we will try to capture this behaviour in a concise metric, along with a strategy that applications can use to compare the metric against memory reclamation costs.

- When free memory is plentiful, the metric should be such that applications are encouraged to expand their footprints, i.e., to populate caches, or (after prolonged memory availability) to redistribute a distributed computation, etc.

- When free memory approaches exhaustion, the metric should signal to applications to stop expanding. If free memory is being rapidly consumed, *and* if

applications are known to take substantial time to react, then this imminent paging warning should happen well in advance of memory pressure. Otherwise, the system can pretend that memory is available until the last moment.

- If memory pressure follows, i.e., if memory allocations continue and free memory is scarce, then the metric should suggest to all elastic applications to consider releasing memory that is cheap to free and later recreate if needed. The metric should provide a common and well-defined reference point for all applications so that they can compare their memory reclamation costs against it, and independently take actions that happen to be in globally increasing order of cost. Thus, *the severity metric should characterize the cost of using memory.*

- Paging out memory to secondary storage should indeed be preferred as an alternative method of reclaiming memory *if and when* applications decide that their memory releasing methods are more expensive than paging. Note that a large volume of paging traffic can back up disk queues and make it more expensive to reclaim memory than an infrequent spell of paging, and this should be accounted for.

- By scaling the severity metric by the applications' nice values before presenting it to them, a niced-down application should be informed that the severity of memory pressure is proportionally higher, and be induced into taking more drastic but voluntary actions. To complement these actions, there should be

a system-level memory reclamation mechanism that enforces this nice-based prioritization even for non-cooperative applications.

4.1 High-level concepts underlying the severity metric

This section provides an overview of the concepts required to understand what the memory management policy is, what cost of a page is defined as, how amortized costs are compared, etc. The following sections delve into formal statements and analyses of these high-level concepts.

4.1.1 Analogy: Water levels (revisited)

To develop some intuition into the precise form that the severity will take to meet the above goals, let us revisit the analogy of water levels from Section 3.3. There are many possible aspects of memory pressure that we could conceivably characterize; the goal of this section is to make it clear which ones are relevant for adaptive memory management.

Consider Figure 4.1. When containers with different volumes and levels of water are hooked up using a narrow tube, water flows from the container with a higher level to one with a lower level, until the overall levels equalize, and so do the pressures at the bottom of each container.

Consider a translation of this analogy to the memory management world. Containers map to applications; water maps to memory. The total volume of water in

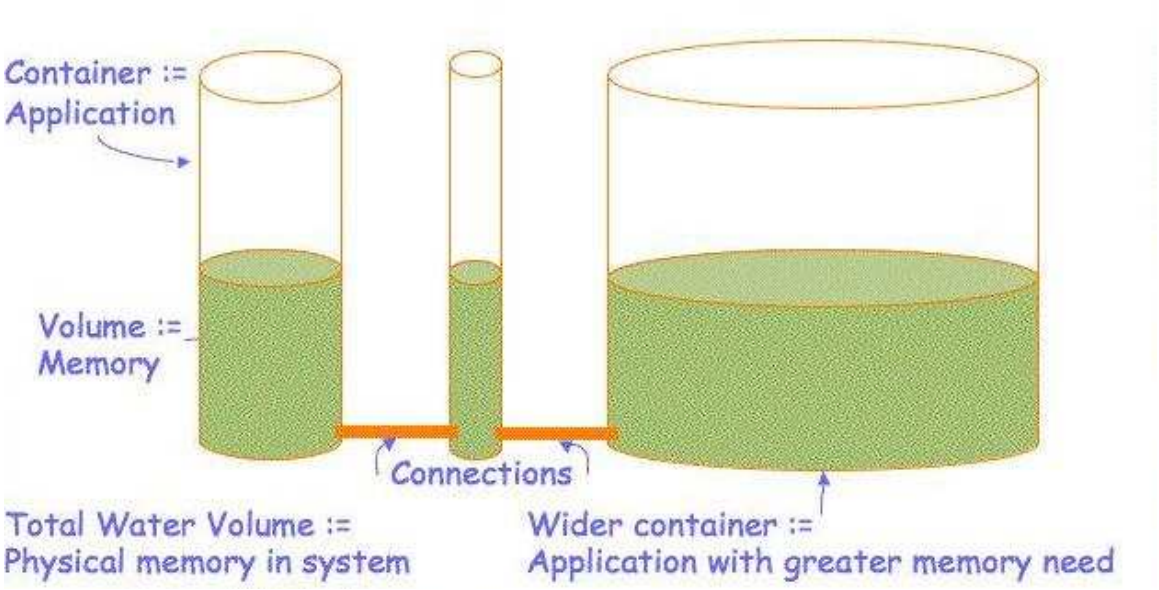


Figure 4.1 : Water levels reaching a natural equlibrium

the system is a constant, as is the total amount of physical memory in an individual computer system. Swap space as an extention to physical memory can be treated separately to make this analogy work.

As we shall see over the next few subsections, this construction shares a number of conceptual analogies to adaptive memory management, and thus allows us to derive intuition into these concepts before we formalize them.

4.1.2 Height of the water column

When the containers are connected by a narrow tube, due to an artifact of our construction, water drains out from the bottom of the higher-level containers. The water molecules at the bottom which leave their containers can be regarded as being the *cheapest* for that container to replace (even though they are identical to any other

water molecule). As more and more water drains out, water that was originally higher and higher up the column also leave. Therefore, this replacement happens in order of increasing cost. Height of a water molecule can be seen as the *value* of the molecule to the container, or alternatively, the *cost incurred by the container if it gets replaced*.

Translating this to memory management, we can see this as replacing memory in increasing order of cost, until these costs are equalized across applications. At this point, the memory pressure experienced by each application also becomes equal. Thus, the policy of replacing pages in order of increasing cost (or value to the application), attempts to achieve our goal of *least-impact-first memory replacement*, where the impact is that incurred by the application due to the memory reclaimed from it.

4.1.3 Cross-sectional area

The volume of a thin cross sectional slice of a container at some height gives us the number of molecules whose cost is equal to that height. The cross-sectional areas of a container therefore become a cost profile of its molecules.

Translating this to the memory regime, it is possible to imagine applications that may access memory in different ways, leading to varied memory cost profiles. For example, an application may (1) possess a linear profile of page access frequencies for the pages in its working set, (2) may have recently touched a large number of pages, and therefore consider consider these pages valuable, (3) could have accessed its footprint a long time ago and thus have many pages that are cheap to release, (4)

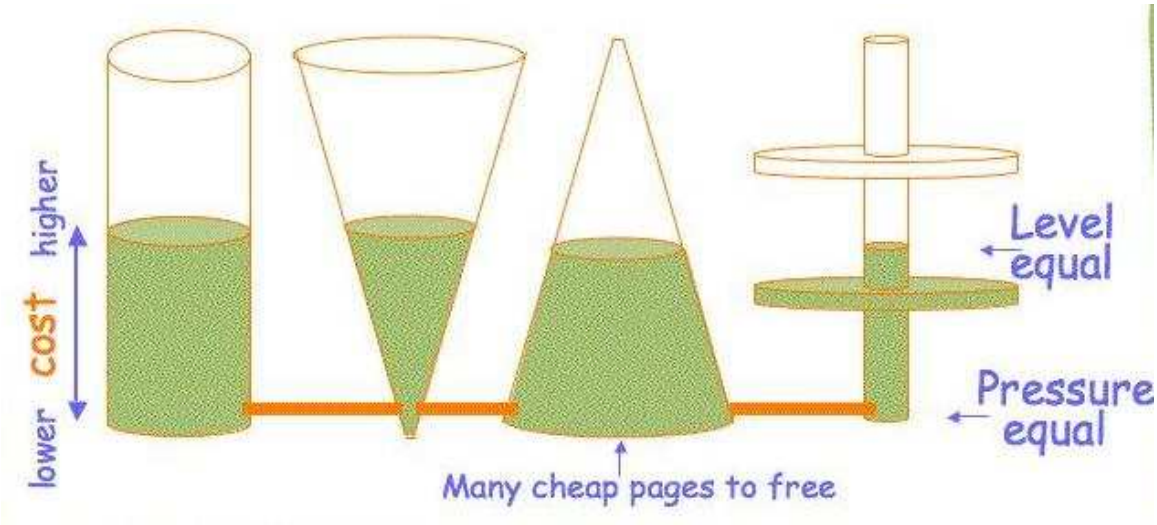


Figure 4.2 : Water levels with various memory cost profiles.

could have a bursty access pattern where it touches all its pages every so often but remains dormant for the remaining time.

Figure 4.2 pictorially depicts these four access patterns as water-level analogies. The take-away message here is that achieving least-impact-first replacement may involve reclaiming different *amounts* of memory from different applications, but in *globally increasing order of cost*.

4.1.4 Cost of a memory page, and the severity metric

Whenever a memory page is being replaced, there are two options: either the application could free up that memory, or the system could page it out. The general idea is to be able to do whichever of these is cheaper for each page.

Based on this dichotomy of choices, we have two methods to estimate the cost of replacing a page: either the application can evaluate the value of the page towards

its own performance, or the system can estimate the cost incurred by paging it out and potentially paging it back in afterwards.

- An application can estimate the value of a page to itself, using whatever criteria it finds appropriate. Given the diversity in application types, this cost function can assume various forms. The general formulation of this cost may be seen as being proportional to the release plus regeneration costs of the page, with the latter de-weighted by its probability of reuse. Furthermore, it may be amortized by the time since last access of this page, to account for active versus inactive pages.

- Independently, the system can estimate the cost of releasing and later regenerating a given page. This cost is proportional to the I/O cost for paging (some function of I/O time and disk queue length), and is amortized by dividing it by the time since last access of this page, which estimates its worth. The system uses this to perform page replacement in amortized cost order, to achieve least-impact-first page replacement.

Conceptually, the severity metric for memory pressure is the *system's estimate of the cost of the cheapest (lowest-cost) page in the system*, or zero if there is abundant free memory available. This information is made available to all interested applications.

4.1.5 Application response by memory adaptation

At any time, an application can derive its own estimate of the cost of *its* cheapest page, and compare it against the system's notion of the same. If the former is lower, then the application knows that the system's overall interest demands that this page be freed up, and paging I/O avoided. If the latter is lower, then the page is valuable enough that it is cheaper to have the system page it out, so that is what happens.

The resultant dynamics of the system are as follows. When there is plenty of free memory in the system, applications adapt to *expand* their footprint to use up most of it, with various benefits obtained by doing so. When free memory in the system decreases, the system pushes back with early warnings, and the cheapest memory adaptations within applications happen. Applications release memory through methods such as garbage collection, freeing unused buffers, etc. If this does not suffice to relieve memory pressure, then the system's estimate of the cost of memory pages increases, and applications become induced to execute more performance sensitive adaptations. They may release more valuable buffers, such as caches of precomputed results or prefetched data. This happens until they reach a stable memory pressure equilibrium, achieved through global least-impact-first memory replacement.

In our implementation, we have the system announce the *age of the oldest page in the system* as the severity metric, along with separate information about paging I/O cost, so that the application can compute the system's notion of amortized page cost in whatever scheme that it finds compatible with its own estimated cost.

In the next section, we formalize the above concepts with equations and analyses.

4.2 Formal definition of the severity metric

We first define the *page age* of a physical memory page to be the time (in seconds) since it was last referenced. We define S_{paging} as the age of the least recently accessed (oldest) non-free page in the system. We will define the severity metric (expressed in time units) based on S_{paging}.

Now we define the system parameters necessary to formulate the severity metric, and in Section 4.3 we explain the intuition behind the values we chose for them. S_{max} and S_{min} are system-defined upper and lower bounds configured for the severity metric. τ is a threshold on the fraction of free memory available, below which the system declares impending memory shortage.

Case 1: When the system has exhausted its free memory: At a time when the system has exhausted all its free memory and may be paging, we define the severity metric for memory pressure to be equal to S_{paging}. Note that a smaller value of the metric depicts a higher level of memory contention*.

Case 2: When the system is about to start paging: If the amount of free memory ($free$) is less than τ, then to signify imminent paging, the severity metric

*Smaller severity values indicate higher pressure. We adopt this convention of reporting the age (rather than say its reciprocal), so that applications can deal with severity in convenient time units, and that our implementation is able to exponentially separate out larger severity values.

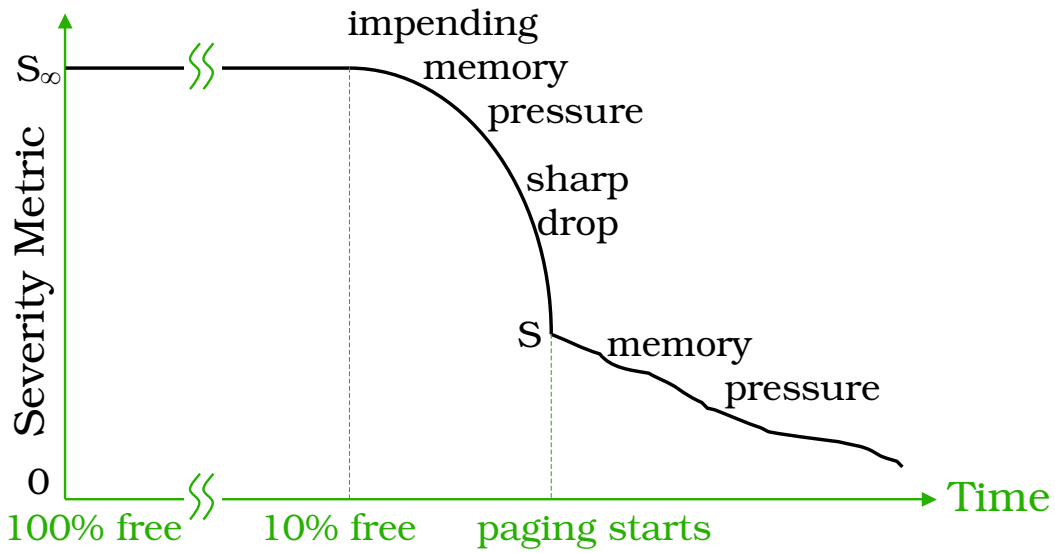


Figure 4.3 : Severity metric dropping as a function of time, when, say, an application steadily allocates memory (schematic).

is geometrically interpolated between S_{max} and S_{paging}. For example, if free memory drops linearly from τ to zero, then the reported severity metric will drop *exponentially* from S_{max} to S_{paging}.

$$S = S_{paging} \left(\frac{S_{max}}{S_{paging}} \right)^{\frac{free}{\tau}} \tag{4.1}$$

Case 3: When free memory is plentiful: When paging is not likely in the near future, the severity metric is set to S_{max}, to declare that there is no memory pressure.

Figure 4.3 schematically depicts the behaviour of the severity metric against time, when there is, say, an application that allocates memory at a steady rate. The metric

remains large (indicating no memory pressure) up until free memory falls to within τ of system memory. Then it exponentially decreases to S_{paging} to indicate imminent shortage and urges applications to take increasingly drastic actions, then as paging starts and grows intense, the metric steadily drops further. This continuity in metric values is intentional, because it encourages adaptations to occur incrementally.

4.2.1 System parameters

We now present the values for these system parameters that we set in our prototype implementation. We do this early on, with the intent of providing some additional insight into the rationale behind these concepts.

We set S_{max} to 1 hour and S_{min} to 5 seconds. The rationale for choosing the former is that we expect there to be little difference in discriminating between physical pages that were accessed more than an hour ago, or similarly application buffers with such low amortized cost. On the other hand, if pages in the system are seen to be churning rapidly, then there is significant memory pressure, and we want to be able to characterize it. The choice of S_{min} is governed by two factors: it needs to be small, so that we can discriminate between pages accessed at different times, but it should not be too small for overhead reasons: measurements in Section 6.6 show that $S_{min} = 5$ is sustainable. Fortunately, 5 seconds is sufficient for our experiments, and probably also for a real system, because S_{min} does not need to be sensitive to the intensity of memory pressure. This can be seen in a situation in which most

physical pages are being rapidly accessed by processes, where it is only useful for page replacement to know which pages were recently accessed, and unlike what most conventional thrashing-oblivious page replacement algorithms attempt, it does not help to know with finer and finer accuracy the particular sequence in which those pages were accessed.

Our prototype also sets τ to 10% of physical memory. This is the threshold at which the system sends out early warnings about memory pressure to applications. Methods of configuring this value and their intrinsic tradeoffs are discussed in Section 5.3.4.

4.3 Use of severity metric

The general idea is for applications to use the metric to balance the level of memory pressure inside and outside its boundary, so that the whole system converges to a state where memory replacements happen on buffers that have the least cost across applications and the file cache (see Figure 4.4).

Let us say an application has a chunk of memory that it can free (with cost C_f in time units), and later require it with probability p and can regenerate (recompute or refetch) it with cost C_r. Let the time since last access of this chunk be T, which the application or the system can maintain. Let the average paging I/O turnaround time be $C_{disk} * l_{diskq}$, where C_{disk} is the average service time for a paging I/O request, and l_{diskq} is the average disk queue length over a recent time interval. Let the severity

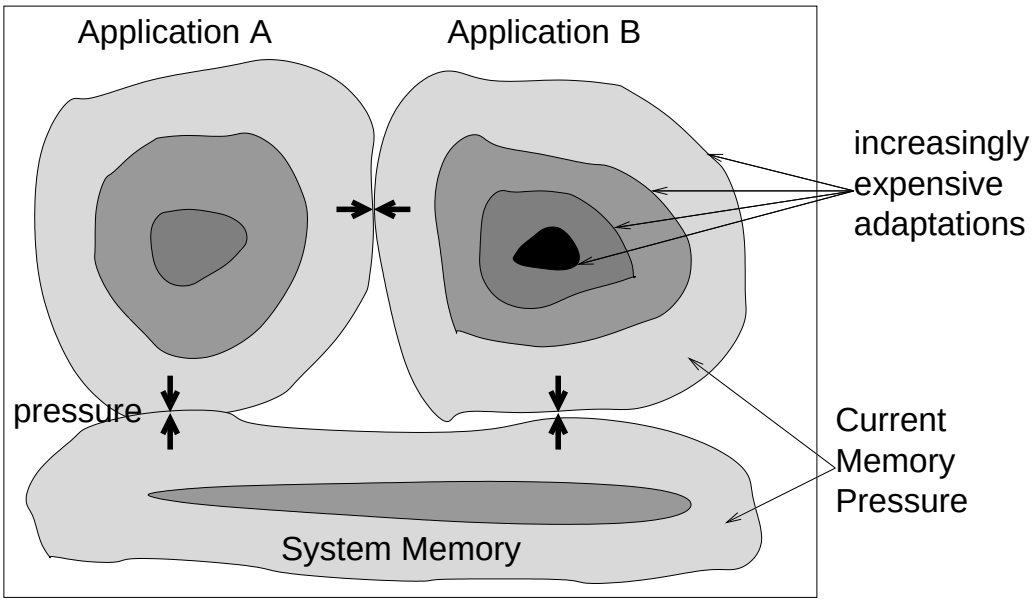


Figure 4.4 : Equalizing memory pressure levels between applications and with the system.

metric S be continuously conveyed to all applications [†].

Assume for now that the system uses a page replacement policy that is based on approximate LRU. If the system is paging, the candidate page for eviction may be chosen from any application, and would have an age of S. Using the LRU principle, we can assume that this page may be accessed again only after approximately as much time in the future. Then all applications can independently calculate the *amortized paging I/O cost* being incurred by the system as

$$C_{paging} = (2 * C_{disk} * l_{diskq})/S \tag{4.2}$$

[†]The system exposes the metric through a system call or a kernel-shared memory location, and additionally uses a new unix signal (SIGMEM) to notify all processes whenever the metric doubles or halves.

They can also compute the amortized memory release and regeneration cost of *one of their own chunks of memory* as

$$C_{app} = (C_f + p * C_r)/T \tag{4.3}$$

If the latter is smaller, then the application knows that it is globally cheaper for it to free this memory chunk than to have the system swap out and later swap in an equivalent number of pages from some application.

Thus, the metric enables paging to serve as a common reference point to compare against application-specific costs of freeing memory. Furthermore, while leaving the inherently application-specific replacement policy to the discretion of the application, this method approximates a global *least-impact-first replacement*, where the physical page or the application buffer that is expected to have the least overall impact on performance is replaced first, to equalize memory pressure everywhere.

When the system is about to start paging (when free memory is below τ), then this region of exponentially decaying severity metric is vital to operate the system in the near-full memory utilization zone without overflowing and paging. The continuous, exponential decreasing characteristic of the metric serves three important purposes:

- it allows applications the time they need to take actions, especially time-consuming ones like garbage collection.

- it conveys the urgency of imminent paging only when free memory is low enough, thereby balancing the needs of early notifications against the desire to operate the system close to full memory usage. Thus it provides the desired safety cushion.

- it helps to avoid taking all memory-releasing actions across the system all at once. Instead, a few of the cheaper and/or time-consuming actions may run first, and may free enough memory to relieve memory pressure and avoid taking more drastic actions.

Thus, the gradual transition of the severity metric from S_{max} to S_{paging} allows for adaptations to be staggered, so that cheaper ones happen first, and may suffice to relieve memory pressure and render the expensive ones unnecessary. This staggering also attempts to minimize the performance impact of numerous simultaneous adaptations. In particular, the CPU activity required to perform the adaptations and the resultant cache thrashing effects can be reduced by triggering adaptations in increasing order of their cost.

We control application adaptations by biasing them by nice values, using the following technique. Rather than reporting the global severity metric identically to all applications, *the system reports the value of the metric scaled by the inverse of the nice value of the notified process.* Thus, a niced-down process is informed that the severity is relatively higher, and is induced into taking more drastic actions. This is explained in greater detail in Section 4.4 and other sections.

Applications, elastic or otherwise, are free to ignore memory pressure notifications and take no action, and there is no explicit isolation between processes or agreements between the system and applications on the manner of releasing memory. If they do cooperate in the interest of system performance, then they do not need to take all the above factors into account – a simplified severity check works in many cases, and some adaptations are either so cheap (e.g., releasing unused memory) or so expensive (e.g., desperate thrashing avoidance measures) that it is trivial to decide when they should happen. However, within limits, it is within applications' interest to cooperate in the manner described above, or else they can be niced down and preferentially paged out by the age-aware page replacement mechanism described in the next chapter.

4.4 Extending "nice" to memory

It is useful to be able to discriminate between applications in terms of how aggressively these applications' memory-releasing actions are taken, and to what extent their pages are preferred for page replacement. Common operating systems take a variety of approaches (with different tradeoffs) for defining an interface to express such differentiation, but the mechanisms for implementing them are largely independent of the scheme itself. Our system inherits the Unix method of per-process *nice values* to allow users to prioritize applications.

We can shift the burden of assigning a reasonable set of nice values from po-

tentially unsophisticated users to a combination of system designers, administrators, programmers, and users. This philosophy leaves ultimate control with users, while presenting a sensible default behavior. For example, a *system designer* can implement a policy that identifies and prioritizes interactive applications over others; these priorities may be designed to depend on interactive lags being exhibited by applications. A *programmer* for a background file sharing application can nice-down its own memory, making it more attractive for eviction; and a *user* can request for say an email application to be left untouched while the rest of the system may be paging.

A crude implementation of memory nicing for page replacement may enable the complete *prioritization* of one process from others, so that the former is effectively pinned to memory until the other processes are fully paged out and their actions fully invoked. However, in a general-purpose operating system, one would typically prefer to assign *weights* to applications (where the weights are derived from the nice value of the process as explained in Chapter 5). Then drastic actions and recently accessed pages (e.g., code, static data) from niced-down processes can compete with mild actions and older pages (e.g., cache objects) from niced-up processes.

Application actions can be niced using the following simple and application-transparent mechanism. Rather than reporting the global severity metric identically to all applications, *the system reports the value of the metric scaled by the inverse of the weight of the notified process.* Thus, a niced-down process is informed that the severity is relatively higher, and is induced into taking more drastic actions.

Likewise, page replacement can also be subject to nicing, as follows. The amortized paging costs for a process may be amplified by linearly scaling down the age of all its pages by the weight of the process. *Page replacement is then performed in order of the resulting scaled ages*, and the globally reported severity metric is redefined as the maximum of the scaled ages of non-free pages. Such weighted page replacement would universally impose itself on all processes, regardless of whether they have been modified to use our API; so this age-based page replacement serves as an effective way to control rogue processes.

So far it seems as if our nicing mechanism only serves to scale the aggressiveness of paging and shrinking by a factor, and that a niced-down but very aggressive process can still cause some damage on a niced-up process. On the contrary, as Section 5.1.2 will show, a feature enabled through our implementation provides for either partial or total isolation of processes, as desired. To our knowledge, this feature is novel to our system.

4.5 Analysis of the severity metric

While there is good intuition for our design, in part using the analogy of water levels, this section analyzes certain aspects of our design to develop a more rigorous and complete understanding.

We first define a model for applications, based on (1) a steady-state representation of a typical in-memory cache, and (2) one whose workload follows a Zipf-like distri-

bution, such as most web proxy caches. Then we present a rigorous quantification of the performance impact of nicing an application by a certain factor.

4.5.1 Application model

Our system operates in a relaxed, best-effort paradigm of informing applications about memory pressure, expecting them to adapt as soon as they can (but without any set deadline), and if they do not, then in the longer term indirectly counteracting with page replacements. We believe that this approach is more suited for real-life applications than schemes like memory partitioning [113] or schemes using economic models [28], but it also makes it relatively much more difficult to analyze application behavior in the general case.

We tackle this conundrum by analyzing our hypotheses within the purview of the following *model* that we define to mimic the behavior of typical in-memory caches. We believe that such applications effectively illustrate the intrinsic properties of our algorithms.

- Let the application have a memory footprint of M, divided into chunks that we will call *pages* numbered 1 through M of equal or unequal sizes. Furthermore, let these pages be numbered in the decreasing order of their frequency of access.

- Let $F(m)$ be a function that describes the frequency of access of the m^{th} page. This is a non-increasing function.

- Let $H(M)$ be the hit rate experienced by the application on pages between 1

through M. Therefore, if the application is performing LFU cache replacement, and is allocated memory M, then $H(M)$ will be its hit rate. Then $H(M) = \sum_1^M F(m)$. This is a function that typically increases fast and then slowly, following the law of diminishing returns.

- Assume that cache misses are much costlier than cache hits, so that throughput T is approximately proportional to $H(M)$.

- Let M_0 be the footprint of the application, so that $H(M_0)$ is equal to the maximum possible throughput T_0 that this cache can deliver given the incident load.

- Assume that requests to the cache are issued independent of each other, so that our model becomes tractable.

- Assume that the application has reached steady state.

In the interest of concreteness, we will also specialize this generic model to a certain type of cache, viz. one whose workload follows Zipf's law, or a variant thereof. Zipf's law states that the relative probability of a request for the i'th most popular page is proportional to $1/i$ [41]. Web proxy cache workloads have been reported to follow Zipf-like distributions [18], with the following characteristics (also depicted in Figure 4.5):

- $F(m) = \Omega/m^\alpha$, as governed by a variant of Zipf's law, where α is a parameter for the distribution with a value less than unity.

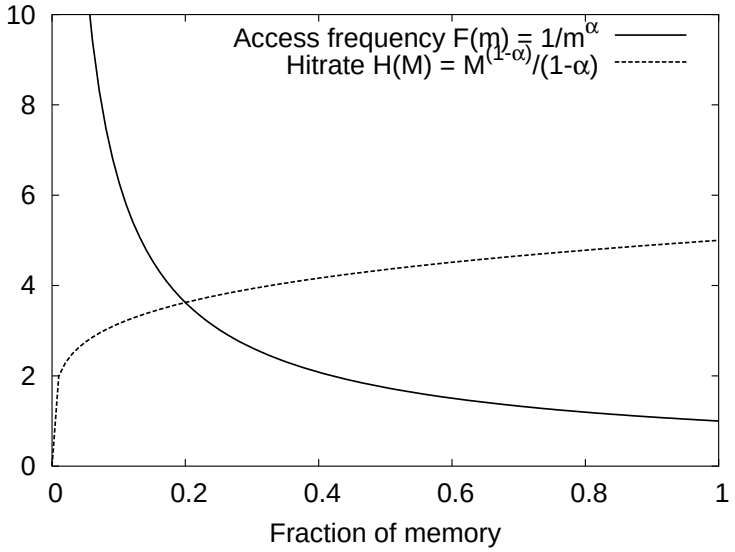


Figure 4.5 : Zipf-like distribution for $\alpha = 0.8$: access frequency $F(m)$ and hitrate $H(M)$

- Summing $F(m)$ from 1 through M, we get $H(M) \approx \Omega M^{1-\alpha}/(1-\alpha)$. The knowledge that $H(M_0)$ is the peak throughput can be used to back-calculate Ω.

- In general, α is larger (centered around 0.8) when the requesting client population is more homogeneous, as in a corporation (DEC) or an academic department (Pisa), than from a diversified user population (UCB) or when it is filtered by first-level proxies (QuestNet and NLANR), and is centered around 0.7 [18].

We define the following additional terminology used during this analysis.

- Let S be the severity of memory pressure, in time units, and

- Let η be the value by which the application is *niced down*, in terms of a factor that the severity metric is scaled down by before presenting it to the application.

The UNIX-style nice value of the process, in the range of -20 to $+20$, can be scaled to compute η in a manner described in Section 5.1.2.

4.5.2 Performance impact of nicing applications

When the system has reached a state of equilibrium, each elastic application would have allocated as much memory as it needs, or until its internal pressure is equal to the system's memory pressure (scaled by the nice value). Consider one such application.

If the oldest page of this application has age T as defined in Section 4.3, then in steady state, its frequency of access would be the least, so this would be the last page when ordered by access frequency. Moreover, even if this particular page was accessed recently, there is very high probability that a page m near the tail of the ordering was not accessed in time $1/F(m)$. Therefore we can assert that:

$$T \approx 1/F(M), \quad \text{where } M \text{ is the number of pages allocated to this application.}$$

Consequently, at equilibrium, Equation 4.2 and Equation 4.3 equate to take the following form, after applying the scaling factor of η for nicing the application.

$$C_{paging} = \frac{2 * C_{disk} * l_{diskq}}{S} \quad \approx \quad C_{app} = \frac{(C_f + p * C_r) * F(M)}{\eta} \tag{4.4}$$

If multiple applications contend for memory, then the above equation enables us to compute the memory partition between them, say to determine if they will share

memory fairly among themselves or whether one application will run the other over.

Identical cache-type applications: We can specialize Equation 4.4 to the case where the applications are identical, and therefore share the same function F(m), the same costs C_f and C_r incurred per access, and the same regeneration probability p. Then if application instance i is niced down by η_i, as a result of which it receives memory allocation M_i (which sum up to the physical memory of the system M_{system}), then we can formulate this:

$$\frac{F(M_1)}{\eta_1} \approx \frac{F(M_2)}{\eta_2} \approx \cdots \tag{4.5}$$

Note that depending on the shape of the $F(m)$ function between 1 and M, these applications may display completely different performance characteristics. Our system has no control over the shape of this curve, nor does it even try to be fair in this performance space between applications with incommensurate workloads.

Two identical cache-type applications with Zipf-like workloads: We now specialize the useful Equation 4.5 further, to consider the example of just two caches that contend for memory, but have Zipf-like workload distributions with the same parameter α. In particular, let the second application not be niced up or down, so $\eta_2 = 1$, and our objective is to compute the performance impact on both applications due to the first application being niced down by a factor of $\eta = \eta_1 > 1$. Combining Equation 4.5 with the model for Zipf-like workloads described in Section 4.5.1, we

get:

$$\frac{\Omega}{\eta {M_1}^{\alpha}} \approx \frac{\Omega}{{M_2}^{\alpha}}, \qquad \text{and}$$

$$M_1 + M_2 = M_1 * (1 + \eta^{1/\alpha}) = M_{system}$$

Therefore, performance degradation on the first application is due to the decrease in its footprint from $(M_{system}/2)$ to M_1. Plugging in the formula for hitrate for Zipf-type workloads (from Section 4.5.1), we get:

$$\text{Performance degradation of app 1} = \frac{H(M_1)}{H(M_{system}/2)} = \left(\frac{2}{1+\eta^{1/\alpha}}\right)^{1-\alpha} \tag{4.6}$$

Correspondingly, the performance improvement for the second application because the first application was niced down by a factor of η is:

$$\text{Performance improvement of app 2} = \frac{H(M_2)}{H(M_{system}/2)} = \left(\frac{2\eta^{1/\alpha}}{1+\eta^{1/\alpha}}\right)^{1-\alpha} \tag{4.7}$$

For example, when $\alpha = 0.8$ and η takes values between 1 and 32, the graphs in Figure 4.6 and Figure 4.7 depict the amount of memory retained by the two Zipf-like applications, and the resulting performance. When $\eta = 1$, the two applications evenly split system memory between themselves and therefore achieve the same performance.

As η is increased, application 1, which is being niced down, suffers more and more. In particular, when niced down by a factor of 32, its performance cuts by half. In contrast, application 2 starts with half the system memory, and increases from there.

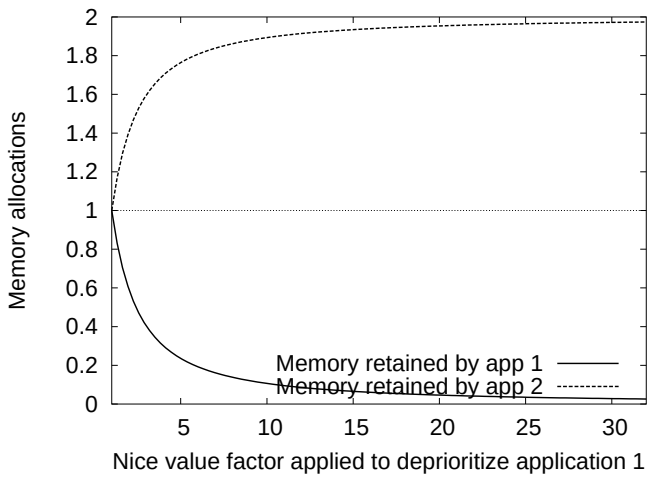


Figure 4.6 : Memory allocated to two Zipf-like applications (with $\alpha = 0.8$) when application 1 is niced down by different amounts.

Even though it rapidly gets most of system memory with increasing η (80% for $\eta = 5$), by the law of diminishing returns, it improves in performance by a relatively small amount (less than 20%). Of course, if these applications happened to have a much larger working set, or if the workload had smaller α (say 0.3), or if there were several such applications contending for memory, then the second cache performance would increase much more as a result of nicing down the first.

4.6 Application programming interface

Conceptually, we want the system to always keep all applications informed about the level of memory pressure in the system, which we have defined based on the age of the oldest page. In practice, the system tracks the age of the page that is the current replacement candidate, and uses this as an approximation to the age

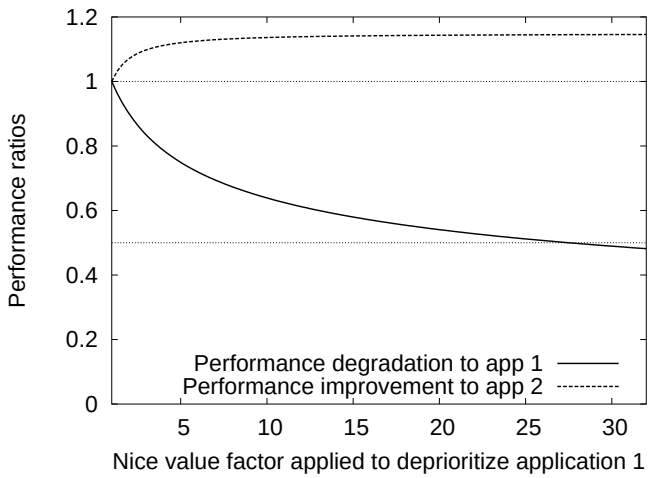


Figure 4.7 : Performance variations experienced by the two Zipf-like applications (with $\alpha = 0.8$) when application 1 is niced down by different amounts.

of the globally oldest non-free page. It makes this metric continuously available to applications through a new system call named `severity`. A new signal named SIGMEM is dispatched to all applications each time the metric significantly changes, such as when it doubles or halves since the last notification. It is also dispatched every $\tau = 1$ second whenever the severity metric is smaller than S_{max}, thereby enabling applications to take periodic actions.

Applications are free to ignore this signal. However, interested applications may implement signal handlers that query the severity metric upon signal receipt, and then potentially decide whether to release or allocate some memory based on logic described shortly. These signal handlers may reside in different layers of software, such as in application code (possibly even in the source code of high level programs), libraries, or language runtimes; in the latter two cases, the actual application may not need to be modified.

There is no need for a special API to express user-specified weights at a process granularity: we simply extend Unix-style per-process CPU `nice` values to specify per-process weights for the memory subsystem. In the case that users wish to have different CPU and memory nice values, our system also provides an overriding system call named *memnice* that sets only the latter.

We provide a resource utilization interface through a `resutil()` system call that reports the percentage utilization of CPU, disk and network interfaces over the past interval of say 10 seconds. Also, an `iocost()` function reports the average time taken for recent paging I/O operations to complete. This includes the disk queueing time for paging requests only if the queue length is more than a threshold (10 in our implementation); this is to remain conservative about our prediction accuracy over the rapidly fluctuating disk queue. These system calls may be used by some applications to derive a better estimate of paging I/O time, and to decide if an adaptation that affects some other resource is justified if that resource is under contention.

4.7 Guidelines for application modifications

As mentioned earlier, the nature and extent of applications' reactions to memory pressure is based on their own discretion and policies. While they may completely ignore the notifications or even maliciously allocate more memory when there is pressure, it would generally be in their own interest to free some memory to avoid paging, and allocate more memory when it is available.

Hence, this chapter provides detailed guidelines that cooperative applications of different types may choose to follow. The following are four types of increasingly drastic actions, where the application may react in the described manner to the severity metric notification, in an attempt to achieve the global least-impact-first replacement described earlier.

(1) When the free page list is short and shrinking, the reported severity metric decreases exponentially towards its target value, and triggers inexpensive memory releasing actions that are taken before paging even starts. For example, garbage collection and releasing malloc-held unused memory are typically cheaper than paging.

The malloc library can cheaply release the free pages it retains, so this action can be programmed to take place whenever the severity is anything below S_{max}, or in practice, whenever there is a notification for any value of severity.

Full garbage collection such as in JVMs is more expensive; moreover, it is usually not possible to determine the cost and yield of garbage collection beforehand (except approximately through statistics). Therefore, we trigger garbage collection only when memory pressure is slightly higher, when the metric is below a constant (10 minutes in our implementation). This will allow cheaper actions to be taken first, and will also ensure that old and potentially less useful pages lying in memory get released or paged out before garbage collection happens. Thus, the JVM would get a larger footprint at a relatively low cost.

(2) If memory releasing actions that are cheaper than paging are unsuccessful in releasing enough memory to avoid paging altogether, then free memory exhausts and paging starts, and the severity metric progressively decreases. Some applications (such as databases) maintain in-memory caches of file data, whose regeneration cost is trivially equal to I/O cost. They can release cache buffers that were accessed less recently than the severity metric, so that the buffers are prevented from unnecessarily being paged out to disk. Also, the system-wide LRU-approximation replacement policy is being naturally extended to application caches, contingent to alternative replacement policies that the application may employ.

In practice, the application may try to match its internal memory pressure with the system's memory pressure, say every time the severity metric doubles or halves, i.e., when SIGMEM notifications are dispatched. In the signal handler, rather than try to calculate the desired size of the cache, the application may enlarge or shrink the cache until the appropriate severity criterion is met. This makes the programming model for the database cache (or the modifications to existing databases) very simple.

(3) Some applications may be able to release memory that is soft state, i.e. it can be regenerated if necessary, and is not backed up by files or in swap. However, this memory may be more expensive to regenerate than to page it out and back in. Such applications may wish to empirically estimate the cost of releasing such

pages and express it in time units. Then our guideline for cache replacement is to compare the amortized time-cost of paging against the amortized time-cost of freeing and regenerating these buffers, and to perform whichever action is cheaper.

For example, a database query cache may be populated with precomputed results that are soft state, but whose regeneration may involve several I/O operations or the use of some other resource such as CPU or network bandwidth. Applications such as Adobe Photoshop may maintain buffers that are the result of expensive image processing operations, and may expect these buffers to be used again in the future. When the disk is relatively inactive (i.e. some paging I/O can be tolerated), it is better to have the system page out these buffers, mark them clean, and reclaim the memory if necessary, rather than have the application lose the memory.

(4) If the system does start thrashing despite all the actions described above, then it may become too unresponsive to remain useful. It becomes vital (for either interactive or server systems) to release memory immediately if possible, and regain stability.

If this happens, the severity metric would drop to a very low value, because recently accessed pages are being evicted. Some applications may be able and willing to react to a very low value by taking drastic actions. They may disable some memory intensive features, or switch to alternate algorithms that use less memory. They may sacrifice responsiveness, functionality, accuracy, or consistency, or may cause the increase in consumption of some other resource – all these towards restoring stability

in an otherwise thrashing system.

For example, Adobe Photoshop may be amenable to free up all the buffers it may have allocated for its filters if the severity metric drops to a few seconds. A web browser may temporarily suspend image or Flash animations and release some memory, and resume it later when the system stops thrashing. A database cache may return stale or approximate results depending on what is acceptable to the application. An image viewer can show lower fidelity images that the user can later sharpen if desired.

Thus, memory adaptations can be effective and progressively drastic if applications cooperate along the guidelines described above.

Chapter 5

Enforcing memory prioritization

This chapter designs the in-kernel page replacement system that is an integral part of adaptive memory management. It permits users to enforce memory prioritization on applications that do not cooperate on receiving memory pressure notifications, thus indirectly providing applications the incentive to cooperate. In the process of achieving this, the system also becomes able to compute and issue notifications of the severity metric.

Having described the design of this system, this chapter defines and provides solutions to the key design challenges we encountered, and discusses issues that arise when deploying this system for real.

5.1 Page replacement system design

This section describes the design of the kernel component of our system, which comprises of page age maintenance, severity calculation and notifications, and age-aware page replacement. We briefly describe these here, since a comparison of page replacement strategies is not the main focus of this dissertation.

5.1.1 Maintaining page ages in bins

Page ages are computed while the OS scans page tables to inspect page reference bits. If the bit on an unreferenced page is found to be set, then the OS places this page in a *page bin* containing pages of age between zero and S_{min}, which is configured to say 5 seconds. Then every 5 seconds, the OS scans these pages, moves unreferenced pages to the second bin with pages of age 5–10 seconds, and moves referenced pages back to the first bin. Pages in the second bin are slightly less likely to be accessed in the next 5 seconds, so the OS scans pages in this bin every 10 seconds. In this manner, the OS maintains bins with exponentially increasing age ranges until S_{max}, and scans pages in each bin with the frequency of the age of the pages in that bin.

These exponentially organized bins (a.k.a. a logarithmic histogram) cover the large age range of 5 seconds through 1 hour using only 10 bins. The compromise is on the accuracy of ages: this scheme maintains ages correct to a factor of 2, and scans it just as frequently. The exponent of the bin index multiplied by S_{min} gives the page age, so the OS does not need to measure timestamps while scanning and also does not store per-page timestamps. A set of such age-based bins is maintained for each process, and pages of that process are stored in these bins.

Now that page ages are being efficiently maintained, the next goal is to select the page with maximum age, if the system wishes to perform LRU-based page replacement. For this purpose, bins of the same index from different processes are *chained* together on one chain per bin index. Selecting a page with the maximum age proceeds

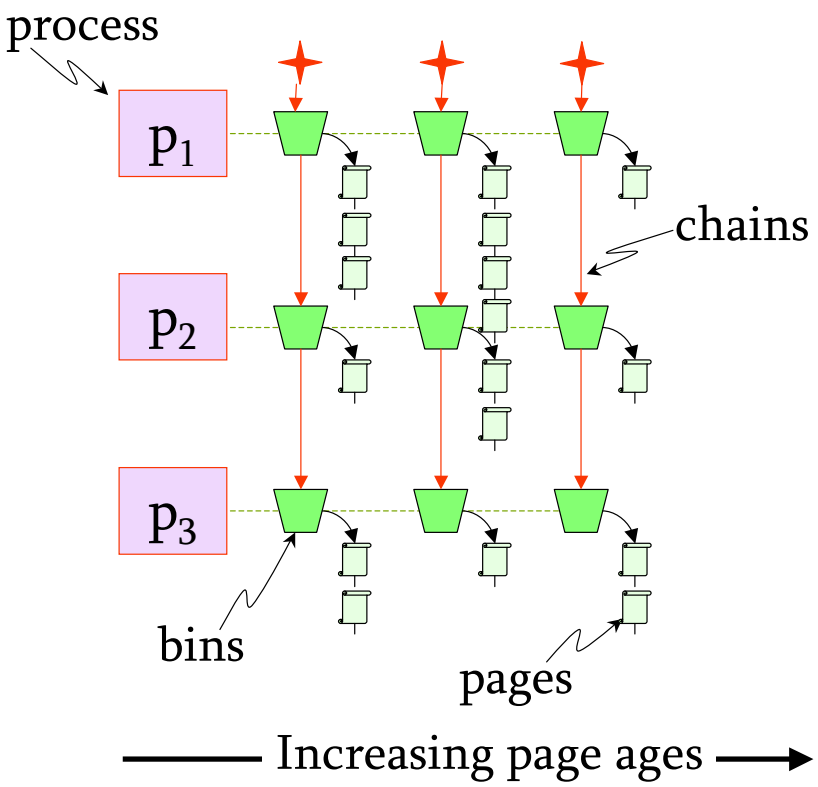


Figure 5.1 : Maintaining pages in age-based bins.

as follows. The system traverses these bin chains from largest index downwards (right to left in Figure 5.1), and for each chain, traverses bins in the chain to encounter lists of pages in that age range. A basic approximation to LRU replacement can evict pages in this order; other replacement policies can be implemented using guidelines provided in the next section.

5.1.2 Nicing processes

The next goal is to nice processes by scaling their page ages by some function of the nice value. For convenience, we internally associate Unix-style CPU nice values (typically in the $[-20, +19]$ range) with the memory subsystem. There needs to be a factor that determines how significant the impact of nicing is; this is S, designated to be the scaling constant (30 in our implementation). We exponentially scale the nice value onto the $[2^{-S/2}, 2^{S/2}]$ range of per-process weights, so that nicing (down) by +19 corresponds to scaling up the ages of the process by $W = 2^{S/2}$. On a general-purpose system, S would typically be a small multiple of the number of bins per process B.

Conceptually, when a process is niced, the system needs to scale the process' page ages by this weight, and then reorder all pages in the system by their scaled ages for page replacement purposes. This can be made efficient by avoiding division and sorting operations, as follows. Adjacent bins in any bin list differ in their age range by a factor of 2. To scale up all of a process' pages by W, we simply have to shift each of its bins $\log W$ chains to the right (refer to Figure 5.2). Observe that the exponent of the absolute index of the bin after this shift times S_{min} is equal to the scaled age of the pages in the bin, which is the *severity* of memory pressure if the page is the replacement candidate. Because nicing only involves detaching a small number of bins per process ($B = 3$ in the figure and 10 in our implementation) from their respective chains and reattaching them in different chains, it is very efficient. This efficiency is central to our design, and is useful for purposes described in the rest

of this section and in Section 5.1.2.

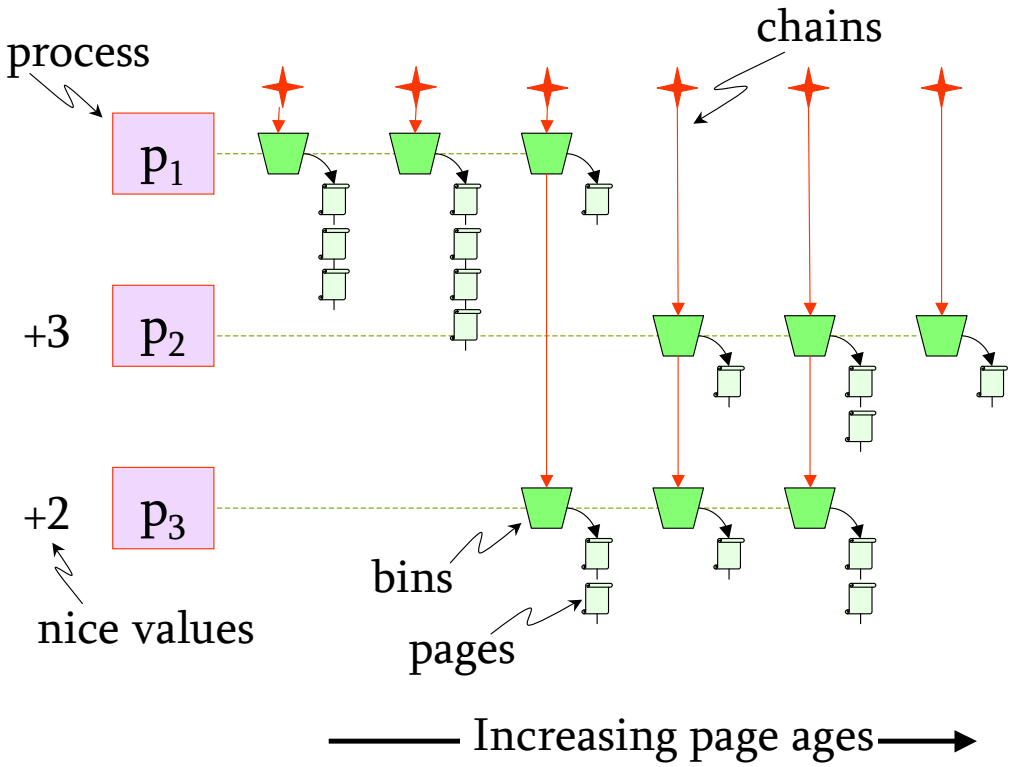


Figure 5.2 : Nicing processes by shifting bins.

5.1.3 Priority classes and overlap

Observe in the figure that the bins of processes p_1 and p_2 do not overlap in their index ranges, due to a combination of system-imposed limits on page ages and a sufficient degree of nicing. These processes have therefore been consigned to different *priority classes*, so that processes in one class cannot induce higher priority class processes to release memory, either by paging or by triggering application actions. There are S/B disjoint priority classes (3 in our implementation), and nicing by $\pm 40 * B/S$ or more

shifts the process to the next priority class.

In contrast, note that bins of processes p_1 and p_3 in the figure overlap to some degree, so the recently accessed pages of p_3 compete with older pages of p_1, achieving the type of overlap usually desired in practice.

5.1.4 Shared and unmapped pages

Page ages are scaled by the nice value of the process associated with the page. However, some pages are shared among multiple processes, whereas others belong to none. The latter consist of unmapped pages, and file cache pages that were accessed by the kernel on behalf of processes that access a file through the read/write interface.

We address the shared page issue by maintaining page ages at the granularity of *virtual page mappings* rather than physical pages. Then we define the scaled page age to be the smallest of the virtual page ages, scaled by the nice values of the respective processes. Age maintenance at the virtual page level is necessary because when a process is niced down (say because it is non-cooperative), its shared pages reflect this new nice value.

We solve the unmapped pages problem by associating such owner-less processes with special sets of *unmapped bins*. Unmapping a page is an indication that the page may not be accessed again, so we *nice down* such pages by assigning them the nice value of the process that unmapped it, biased by a constant (1 in our implementation). This approach relates to FreeBSD's technique of deactivating pages that it considers

unlikely to be accessed again. Furthermore, our approach is more flexible in terms of being able to arbitrarily adjust the bias applied to such pages.

5.2 Uses of efficient nicing

Age nicing provides a versatile framework for two techniques briefly described here.

(a) Nicing-down clean pages

Clean pages are less expensive to evict than dirty pages, since they can be released without first writing them to disk; their I/O cost is only that of paging them in if accessed later. To prefer clean pages for eviction, and to report a severity value to applications that reflects the cheaper cost of clean pages, we scale up the ages of clean pages by a factor of two. It is also possible to dynamically adjust this scaling factor to depend on the amount of transient disk contention. In practice, we achieve this scaling by associating clean pages with a separate set of bins that are shifted one position to the right.

(b) Specialized page replacement policies

This chapter has, so far, described age-aware page replacement using an LRU-based policy. However, our system is fundamentally not tied down to any specific replacement policy. For example, it is possible to easily bias page ages by some function of the *frequency* of page accesses. This achieves a partial least-frequently-accessed first policy, similar to activation counts in FreeBSD and page aging in Linux [74], but with

finer control on the amount of bias. Again, efficient nicing is crucial for implementing this.

To summarize, our proposed age-based memory management substrate facilitates system programmers to implement a wide range of policies, in a more flexible manner than current OSs typically allow.

5.3 Design challenges

We present four technical challenges that we encountered while building the system, along with corresponding solution approaches.

5.3.1 Priority inversion

When a process is niced down, its pages will be preferred for replacement even when they have been more recently accessed than other pages in the system. This may increase the total paging I/O, and may induce some extent of priority inversion by degrading the performance of niced-up processes (as a result of being niced-up).

Solution: Nice for memory is most effective in combination with a proportional disk scheduler such as Cello [98] or YFQ [20], so that nice values can be extended into the disk subsystem. These schedulers allow the user to ensure that disk service is provided to processes in different proportions, which the user may set to match the prioritization in the memory management system. With this scheme in place, the priority inversion problem would not manifest itself, as the high priority process

cannot get blocked in the disk subsystem by a low priority process.

Anticipatory disk scheduling, proposed in this dissertation, is a necessary part of implementing nice values in the disk subsystem, because of the following reason. Implementing priorities in the disk subsystem has limitations because disk scheduling is non-preemptible, and a high priority process may see its disk request wait behind the currently scheduled request to finish receiving service [96]. One such incorrectly prioritized request causes no harm, but this could conceivably happen for every single request from the high-priority application (especially if it issues a synchronous stream of requests). This could result in priority inversion despite using a proportional disk scheduler. Anticipatory scheduling eliminates this problem by sometimes forcing the scheduer to wait for subsequent requests from the high priority process before scheduling the next request (see Chapter 8).

5.3.2 Internal fragmentation

Application buffers may be smaller than physical pages. If some buffers on a page were recently accessed while most others are old, then the application may not wish to release such pages when notified of memory pressure. Coupled with the fact that it may have initially allocated a large amount of memory, this application may seem like a memory hog.

Solution: The application may enforce that its buffers be a multiple of page size (such as in the MySQL query cache); otherwise it may calculate buffer ages as

the *average age* of the buffers in the container page, to allow for fair comparisons against the severity metric. It may also be more aggressive about shrinking if it detects internal fragmentation, or perform some kind of memory compaction (such as using a copying garbage collector) to reduce fragmentation. If none of these options are feasible, then the application will have to pay the price of using more memory pages than its actual memory allocation, and being a memory hog.

5.3.3 Double paging

The very act of an application releasing memory buffers may cause it to inadvertently page in some of that memory. For instance, a garbage collector may need to examine memory pages (possibly after paging them in) before deciding whether they are garbage.

Solution: Applications that may face this issue may consider using the *mincore* system call to determine if a set of pages is in main memory or backing store, and to try to account for the potential page-in cost before accessing or releasing the memory. They could also use *madvise* or *mlock* to pin some of their control structures in memory.

Such a check may not always be feasible; for instance, a garbage collector may be forced to access a lot of data just before collection, precisely at the time of memory pressure. The solution to this is domain specific; for instance, in the garbage collection case, a partial solution to this is to perform generational (partial) or full

collections depending on the distribution of page ages in different bins. If this bin count information were to be exported from the kernel to applications, then in the case that there are practically all pages in the younger and unpaged bins, it is possible to perform a full collection without paging. However, if pages are distributed over bins of all ages in some fashion, then a collection of the younger generation of objects may obtain locality benefits [11]. We have not explored this idea further in this thesis, since it depends on intricacies of the garbage collector implementation [107].

5.3.4 Memory under-utilization *vs.* the risk of thrashing

In Section 4.2.1, we set τ to 10% of total physical memory. This is the threshold that free memory may drop to, when the system sends out early notifications of memory pressure. This induces applications to perform cheap adaptations and raise free memory to be above 10%. If these adaptations did not happen or were not sufficient, then free memory dips below 10%, and the system sends out successive notifications, exponentially increasing in intensity, until applications that can react do so. Thus in steady state, the system tries to converge global memory usage to an amount that is a function of both τ and the expensiveness of adaptations in the applications at that time.

There is always a chance that an application allocates memory too fast for others to react, and causes the system to thrash. Note that this is no worse than conventional operating systems, which usually have no control over the rate of memory allocation.

This gets worse if τ is smaller, but even with a large value of τ, some risk remains.

If τ is set to be too large, then the system under-utilizes memory.

Solutions: We set τ to a value (like 10% of system memory) such that it is reasonably uncommon for real applications to legitimately need to allocate more than this amount within a period of time that does not give a chance to applications to react (e.g., 100ms, obtained from measurements over our test applications). To protect against the case in which this does happen (e.g., when a large application like Adobe Photoshop starts up), we propose the following three solutions as potential extentions to our current implementation.

(1) One can impose a *throttling mechanism* on applications that have requested memory too quickly. This can simply involve delaying the allocator by the amount of time that it takes to schedule other processes that have a chance of releasing memory. This way, by hooking into the CPU scheduler, the system can make sure that memory adaptations get their chance.

Furthermore, the system can explicitly throttle rapid, large memory allocations to make sure they do not cause the system to thrash. There are policy issues surrounding this that are best left to be resolved by the designer of the specific operating system based on usage context.

This solution can remove the danger of thrashing, and allow the system to decrease τ and improve memory utilization. The tradeoff has shifted to limiting memory utilization versus limiting the maximum allowed rate of allocation, and leaning towards

the latter seems reasonable.

(2) Another solution to this problem is to adjust τ dynamically, say in the range of 2% to 20%, depending on how much variance in memory usage that the system observes among its applications, and whether they need enough memory to operate close to full memory utilization, at the risk of paging.

(3) Finally, we can use the mechanisms we propose in this thesis to automatically nice-down rapid memory allocators and potential memory hogs, perhaps even automatically. Then, when free memory is plentiful, these nice values will be ineffective. When free memory drops, the nice value ensures that memory hogging applications or rapid memory allocators mostly or completely replace memory from their own footprints, and do not affect other applications.

Chapter 6

Evaluation of Adaptive Memory Management

This chapter evaluates our prototype implementation of adaptive memory management, and demonstrates its effectiveness and shortcomings.

We implemented a prototype of this memory management system in the FreeBSD-4.3 operating system. We replaced the page daemon by our age-aware design, and added a simple API for querying the severity metric and delivering SIGMEMs. We have conducted an experimental evaluation in various high-level application environments (database, JVM, web server, malloc library) to develop a better understanding of how the system behaves in practice. We also performed experiments with a modified Mozilla browser (not presented here); we were able to demonstrate that the browser can clear its memory cache when notified of memory pressure, and avoid or reduce paging.

Enroute we describe the modifications that we made to different applications, which can be broadly stated as some subset of these: (1) enabling a means of resizing the application, (2) tracking the expense associated with freeing and later expanding chunks of memory, and their latest access times and frequency of access (the latter for estimating if they may be accessed again), (3) implementing a SIGMEM handler, and (4) resizing the application by an appropriate amount using a suitable policy,

and releasing or allocating memory as needed.

6.1 Experimental hardware

Our experimental harware is a 1.7GHz Pentium IV with 256MB of main memory and a 20GB IDE disk. In the overhead microbenchmark in Section 6.6, we also use a system with 1GB of RAM to characterize overhead as the size of main memory scales.

6.2 Evaluation on database workloads

This suite of experiments is perhaps our most complex one, and serves to evaluate various aspects of the system, like adaptation on memory shortage, reaction time, impact on the file cache, application replacement policies, etc.

6.2.1 Problem statement

It is a well-known problem that database applications lack a reliable mechanism to know how much memory to reasonably request from the system, leading to either overallocation and paging, or memory under-utilization. Therefore major database systems require tedious, manual configuration of their memory settings from time to time, to adjust to workload and other applications.

It is also well-known that databases do not cooperate well with other large applications running on the same system. Not knowing how to share memory between

these applications is one major reason. As a common workaround, database servers are generally run on separate machines from other applications.

6.2.2 Summary of findings

We modified the MySQL database to support memory adaptation, and ran the ATIS (Airline Transaction Information System) benchmark, described in detail in Section 6.2.4.

- Even without memory adaptation, we observed a 20% speedup in the benchmark as a result of using memory liberally, by allocating most of the 256MB in the system plus some of its swap memory, rather than the default 100MB that MySQL was initially configured with.

- We experienced a speedup of 14% over the original benchmark by adapting to memory pressure and operating at 90% memory utilization. The overhead of adaptation and memory underutilization is therefore 6%, but the benefit is being able to accommodate memory allocation bursts, as follows.

- A memory intensive application was injected into the above system. Without memory adaptations, the database starts thrashing and eventually dies. With memory adaptations, the database degrades gracefully to 58% of its original performance, and the system is relatively robust to memory usage fluctuations.

We ran the ATIS benchmark alongside an Apache web server on a NASA trace workload, to see the impact of one on the other when they run in conjunction.

- Without memory adaptations, the database pushed the web server out of physical memory, starving the latter.
- With our system, the two applications divided physical memory between themselves according to their respective needs; the database got 150MB and the web server got 50MB.
- The database ran at 82% of its standalone performance and the web server ran at 69% of its. Each suffered only a moderate hit due to the other.

Details of these experiments follow.

6.2.3 MySQL modifications

We modified MySQL v4.0.10-gamma to incorporate memory adaptations. MySQL maintains a query cache of constant, preconfigured size, which we retrofitted to automatically resize on SIGMEM notifications. We then query the severity metric, decide the amortized cost threshold to stop freeing memory at, and limit the amount of memory to free in each shot to an estimated 10% of system memory (this is evaluated below). We then free whole pages (rather than query cache buffers) in order of increasing amortized cost until this threshold is reached (we present an experiment that compares this against LRU replacement). We release these pages to the system using batched invocations of the `madvise` system call for other applications to use. To facilitate replacement by amortized cost, we (1) measure and distribute the evaluation time of each query among the pages that back the cache buffer; (2) register the

time since latest access of each buffer (this is inexpensive and does not need system support), and (3) note their frequency of access to guess the probability of future accesses. We have the application bin its cache buffer pages by approximate amortized cost computed using the equation in Section 4.3, and we progressively move them between bins as they age.

These changes require a patch of 300 lines of C++ code in the MySQL query cache, and they incur negligible runtime overhead.

6.2.4 Workload Characteristics

We used the ATIS (Airline Transaction Information System) benchmark to evaluate memory adaptations in the database system. This benchmark populates entries into a database, and performs many iterations of 43 different types of SELECT queries on it. Some of these query types proved to be an order of magnitude more expensive than others, as may be the case on many real-life database systems. As a result, this database benchmark is memory intensive, and its performance is sensitive to the size of the database cache, the cache replacement policy, etc.

This property was exhibited to a lesser degree by the standard TPC benchmarks; for example, TPC-C lays emphasis on transactions rather than caching behavior, TPC-B has few query types and stresses cache management to a smaller extent. Therefore, we used the ATIS benchmark to evaluate memory adaptations.

We configured ATIS to populate transaction entries into a database of 200MB

footprint, which did not completely fit in RAM on our experimental machine. We run 10 iterations of each of 1000 distinct versions of 43 different SELECT query types on it. These 43 types, coded into the ATIS benchmark, have widely different evaluation costs plotted in the histogram in Figure 6.2. The following are the performance characteristics of this workload on our system.

Observe the 20.5% performance increase exhibited by this workload while using a liberal 300MB memory cache size over being configured at a conservative 100MB. Another interesting take-away is that 300MB is larger than our system memory, and yet a small amount of paging is actually beneficial to performance. Thus, paging in small quantities is not always undesirable, and subsequent experiments show how our modifications to the database understand this and free memory only when memory pressure grows beyond the initial paging point.

Cache setting	Runtime	Hit rate	Footprint
300MB	28.8 min	90.2%	195MB + 26% swap
170MB	34.1 min	63.2%	183MB
100MB	34.7 min	41.0%	109MB
0	46.7 min	0%	3MB

We considered using the more well-known TPC-B (scattered random access transactions with all-or-nothing cache reuse) or TPC-W (OLTP benchmark operating out of a small working set) instead, but ATIS has a gently falling off working set profile that demonstrates the use of increased memory footprints (see table above). Commercial databases possess such characteristics, which is what makes cache size tuning important [111, 14].

6.2.5 Database benchmark and a memory hog

Consider Figure 6.1 depicting system dynamics for the above database experiment, with a memory hog introduced at 10 minutes. The hog, in this experiment and in future ones, represents an unexpected burst of memory consumption (perhaps in a legitimate, non-malicious application that may need the memory) which can drive the system to page or even thrash. Being aware of this possibility, in current systems, the database needs to be cautious and conservative about using memory.

The solid curve plots the severity metric, and the dashed one tracks the database cache size as a function of time. Generally speaking, note how the latter follows the former in interesting regions – this is a sign of prompt adaptations. To make the graph easy to read, we configured S_{max} to 100,000 seconds rather than 3600, and saw no effect on the results of this experiment.

Initially, the system has 200MB memory available for applications, and the rest is in active, wired, inactive, file cache, etc. memory. For the first four minutes, the database steadily allocates 180MB and consumes all but $\tau = 10\%$ of system memory, and the severity metric remains at S_{max} to indicate no pressure. Then the system notices that memory shortage is imminent.

Then for the next two minutes, before it completely runs out of memory, the severity metric exponentially decreases down to 512 seconds, which is the age of the oldest page in the system at this time (rather than remain at 100,000 at the dotted line). Memory pressure notifications are sent out, and the database reacts within

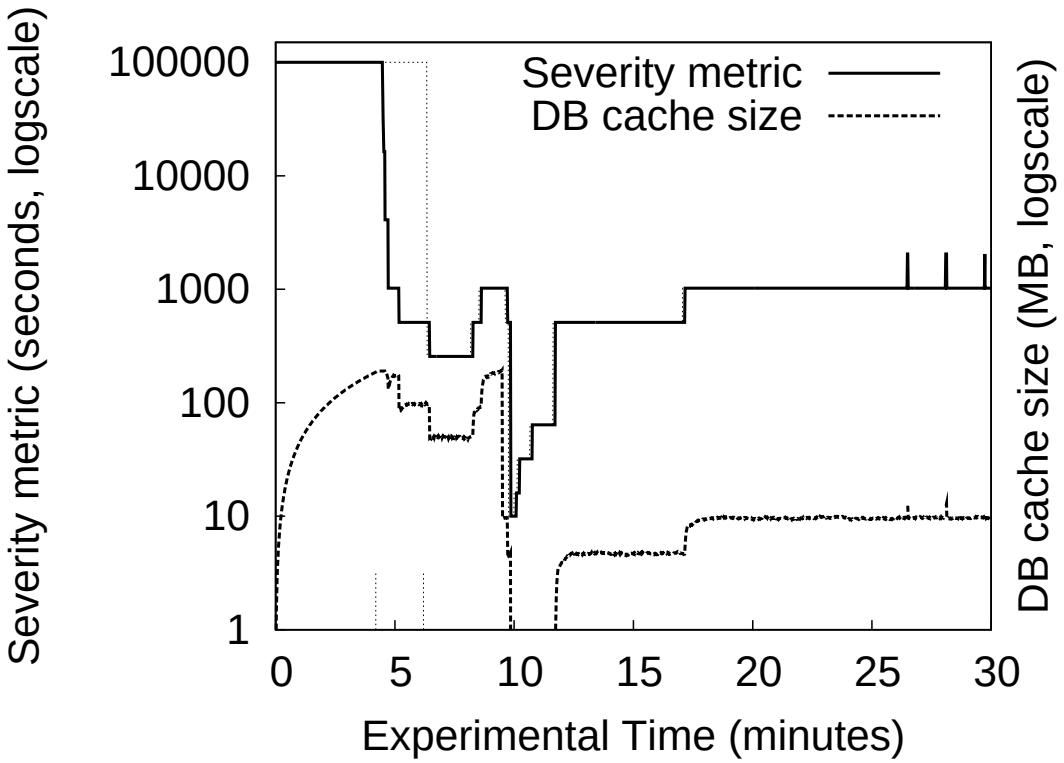


Figure 6.1 : Severity metric dynamics and database cache size reaction.

100ms to this pressure by releasing some its memory in amortized cost order, from cheaply regeneratable buffers. The first replaced buffer has an amortized cost of .877ms / 238s = 4e-6, which is very small in comparison to some of the database's most expensive buffers with an amortized cost of 0.1, or to typical amortized paging costs of 1e-3 to 1e-2.

Between 6 and 10 minutes into the experiment, the system reaches a steady state. There is some relief in memory pressure at 7-8 minutes as a result of the page daemon laundering some dirty memory that now becomes cheap enough to reclaim, and the database increases its cache to use this memory. Observe how the database considers

its cache memory to be worth more than the penalty due to some scattered paging, and indeed, we will show how this plays an important role in the database performance.

At 10 minutes, we introduce a memory hog that causes a burst of severe paging activity. Memory pressure rises, and the severity metric immediately drops. Almost immediately, the database reacts by freeing essentially all its cache, to be fair to the memory hog. Again, we measured the reaction trigger time (signaling latency, application signal handling time, etc.) to be mean 8ms and max 88ms, and it finishes releasing its cache memory in 0.4 seconds.

For comparison purposes, we ran this experiment on the original system, with the database configured to use a 200MB cache. The system reached a steady state of full cache population, then we introduced the memory hog. Since the database cache size did not adapt, its footprint in memory reduced when the database got progressively paged out (and paged in when subsequently accessed), the disk queue lengths suddenly increased to over 40 due to the pageout traffic, starving the page-in requests, and the experiment did not complete in reasonable time.

6.2.6 Database performance

When the database uses all of system memory instead of the conservative 100MB, its performance is higher by about 20%, showing that this workload can make nontrivial use of more than 100MB of memory.

When free memory falls below $\tau = 10\%$, the database frees some cheap buffers in

lieu of paging them out, and tries to operate the system at 90% utilization (except for expensive buffers that are indeed paged out). This causes a performance degradation of 6% (running time of 30.6 min compared to 28.8 min in the original system), which is the price we pay for (1) the ability to accommodate future memory allocation bursts of up to 10% of system memory, and (2) the advantage of not needing to manually configure the database cache size, but to automatically use as much memory as possible.

In the above experiment, however, the introduction of the memory hog delayed the run and caused it to complete in 40.9 min, whereas the corresponding experiment in the original system started thrashing and did not finish at all.

As we developed this system, we encountered interesting issues that caused large performance degradations until they were corrected. Some lessons we learned are:

(1) Applications should release memory in batches of a few MB, then recheck the severity metric to see if either this or any other memory release has relieved memory pressure. Not doing so can unnecessarily release too many of its buffers, which we found to cause degradations of up to 47%.

(2) Least amortized cost first works better as an application cache replacement policy than LRU. We performed a controlled experiment with a fixed cache size of 170MB, and triggered replacement upon memory pressure notifications, and varied the replacement policy. We observed runtimes of 40.9 min with LRU, 37.4 min with LRU augmented with a "lookahead" that checks if there are any low amortized cost

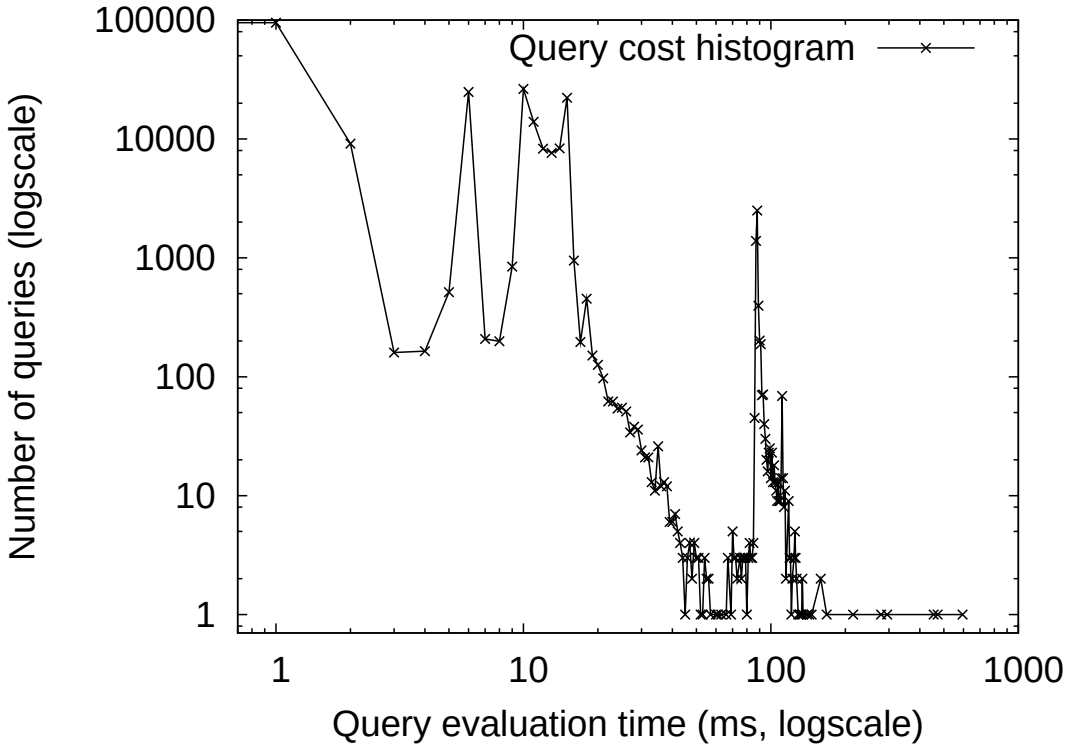


Figure 6.2 : Query evaluation cost histogram.

buffers just ahead of the replacement position in the list, and 34.1 min with amortized cost first replacement. The reason the latter policy performs 20% better can be understood from the query cost distribution in Figure 6.2. There are many buffers that are cheap to recreate, some medium-cost buffers and a few expensive ones. It is performance critical that the latter be retained in cache for longer time after they are accessed.

6.2.7 Database benchmark and its impact on the file cache

One penalty introduced by our system is file cache depletion, and raises the somewhat intangible comparison of the benefit in applications using more memory against detriment caused by lack of this memory from the file cache or from other applications, and in general, how complex systems would behave. We claim that if the database enlarges its cache, it is because it believes that its memory is worth more than the file cache pages. This belief could be incorrect if, say, the database cache buffers are never accessed again.

We attempt to empirically quantify this by comparing the benefits in a database against the cost of file cache misses in a co-located web server. Traditionally, these are not co-located because databases tend to push web servers out of memory; with memory adaptations the database can choose to be cooperative.

We set up an apache web server on a NASA web server trace, serving out of a warm cache, and measured its peak throughput with and without the database running alongside. When running standalone, the web server achieves 977 req/s with a 220MB memory footprint. When we run the ATIS database benchmark alongside, the latter only grows to occupy 150MB and leaves 50MB for the web server, because the web server is frequently accessing most of its pages. The database shows a runtime of 35.1 min (slowed down partly due to CPU contention) and the web server slows to a throughput of 680 req/s.

Thus we show that the file cache suffers only a moderate hit due to the database,

and vice versa, because our system enables them to compete for memory. In particular, the file cache does not get completely wiped out by a memory-hungry database that is oblivious of other memory consumers in the system. Our severity metric allows the database to gauge memory pressure without knowing the identities of other applications or the file cache.

6.3 Garbage Collection

This section evaluates the performance problems that virtual machines face by not being aware of how much memory they should be managing and garbage collecting over, and measures the improvements if the operating system informs them about memory pressure, thereby providing them with this information.

6.3.1 Problem statement

It is well known that getting high performance out of garbage collected language runtimes requires that they use sufficient memory that garbage collections are infrequent. This is often achieved with the workaround of isolating the virtual machine so that it can use almost all of system memory. If the virtual machine needs to be colocated with other applications, then either it may be configured with a fixed amount of memory (disregarding any variations in other applications' memory needs), or it may compete with other applications for memory – either way, there is fundamental tension between assigning a large heap for the virtual machine versus allocating

sufficient memory to other applications.

6.3.2 Summary of findings

We ran a memory-intensive simulation application on the Sun Java virtual machine (version 1.4.1_02), with two different parameters. We were able to achieve a reasonable balance between memory allocated to the JVM versus memory allocated to other applications.

If the JVM uses a heap size of 200MB rather than 50MB, then the simulation exhibits a factor of 4 or 5 less in relative garbage collection overhead, which drops from 25%–80% of its total runtime to 5%–10%. Thus, there is benefit to using a large memory footprint, which adaptive memory management realizes.

Then we introduced a memory intensive application that consumed 90MB. The original system degraded by 150% in one case and thrashed in the other. With adaptive memory management, the performance degradation was graceful, and limited to 25% and 55% in the two cases.

6.3.3 Elasticity

First we explore the elasticity in Java VMs, to measure the extent to which a large heap reduces garbage collection overhead and improves performance.

We used a simulation experiment based on the Scribe peer-to-peer multicast system implemented on FreePastry [26]. We wrote a Scribe application where 10,000 Scribe nodes send 100KB messages to other nodes; each node maintains a local buffer

of up to ten messages to simulate a streaming video multicast scenario. We also experimented with a message size of 1MB, to exercise a more memory intensive situation.

Figure 6.3 depicts the fraction of time that the Java VM spends garbage collecting for different heap sizes, starting with the minimum required for the application to run (22MB and 43MB respectively for the 100KB and 1MB cases). The graph shows that the application is elastic in that it can trade collection overhead for memory footprint. Providing the two simulations a large heap size of 220MB decreases the garbage collection overhead by factors of 6.7 and 4.4 respectively, with corresponding speedups of 2.1 and 4.4.

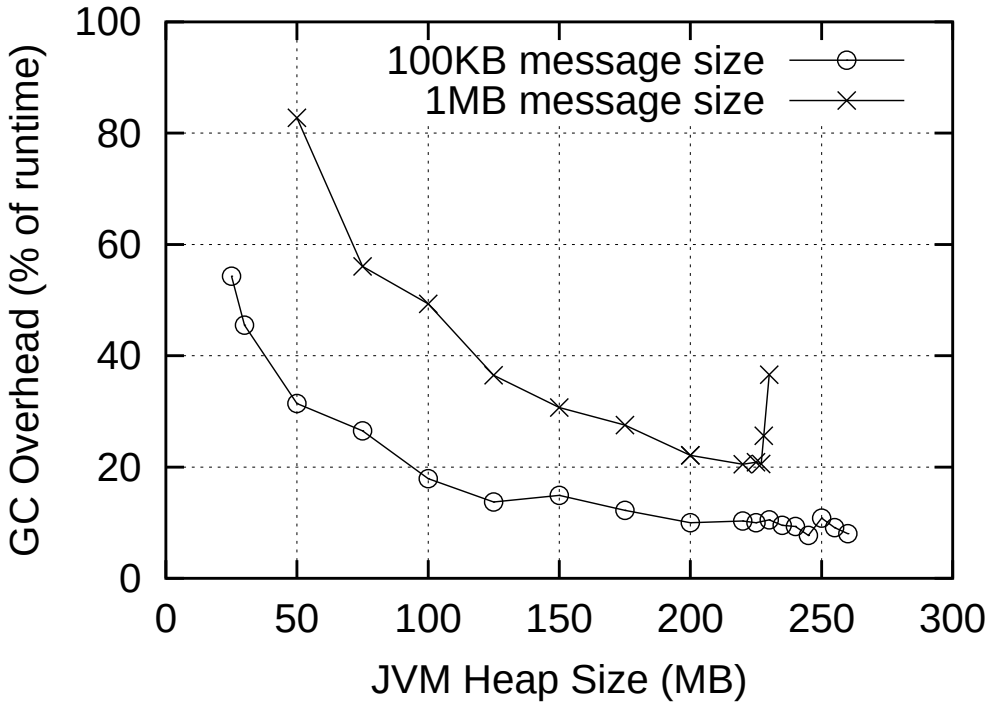


Figure 6.3 : Fraction of GC time.

If the heap size is set to exceed 220MB, then it overflows the free memory available for applications on the 256MB machine. Then somewhat surprisingly, the 100KB simulation does *not* thrash: this is because this application continuously generates independent garbage objects that are easily collectible; moreover, the periodic generational garbage collection of the young heap as performed by the Sun JVM suffices to match this rate of garbage generation. On the other hand, the 1MB case generates garbage too fast for the periodic collection, so it thrashes, causing the spike in the graph. The table in Figure 6.4 reflects this spike in running time for the sensitive 1MB simulation against different values of the JVM heap size, showing that for some applications thrashing can be a performance disaster.

Memory consumption	Runtime
220 MB	240.7 sec
225 MB	249.3 sec
227 MB	255.7 sec
228 MB	506.8 sec
230 MB	817.7 sec
231 MB	never completes

Figure 6.4 : Runtime spike in the 1MB case due to thrashing.

6.3.4 Severity-based adaptations

Now we show how the application can react to memory severity notifications from the system. One could retrofit this adaptation into the middleware layer, viz. the Java virtual machine, and not have to modify Java programs at all. This would be the preferred choice for a real deployment.

However, we did not have access to the Java virtual machine code, so for our experiments we achieved the same effect by instrumenting the Scribe application with a signal handler for severity notifications. This signal handler is coded within a JNI module [57], which exports a simple registration scheme to the Java program. On receipt of SIGMEM, the JNI method calls the `severity` system call, and passes this value to the Java-based handler. This handler simply calls `System.gc()` if the severity metric is less than a constant, set to 10 minutes in our implementation. Thus, in times of memory pressure, some old pages in the system get evicted first, and then garbage collection occurs. If the severity remains high, then the notification arrives periodically every second, thus triggering garbage collection sufficiently often.

We set the JVM heap size to a very large value (10GB), and allowed the severity notifications to control it exclusively through garbage collection. Under normal execution, the heap grew till 220MB and then periodic garbage collection maintained the system close to full memory utilization. The entire experiment runs in less than 10 minutes, so when the severity metric for the JVM pages is compared against garbage collection, the latter gets preferred over paging out parts of the JVM. This eliminates the need for manual heap size configuration.

However, operating close to available RAM may be dangerous if another application suddenly consumed a lot of memory. To study this, we run a memory hog that consumes 90MB of memory. Without adaptations, the 100KB simulation degrades performance by 150% (runtimes depicted in Figure 6.5), whereas the 1MB simulation

thrashes and never completes. This (and some of the noise in this experiment) cause runtimes to decrease and then increase when the JVM is allocated more and more memory.

With the severity notifications and the garbage collection handler, the JVM reacts to the memory hog by gracefully decreasing its footprint until both applications are accommodated in RAM. As a result of using a smaller heap, the two simulations suffer a slowdown of only 25% and 55%, and the system does not thrash.

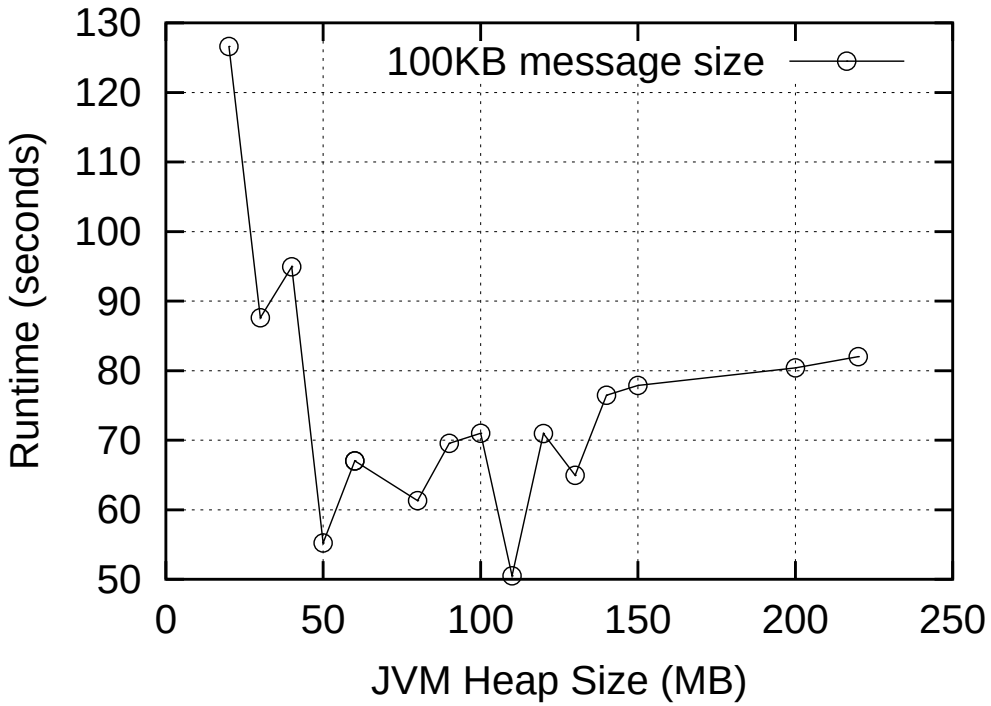


Figure 6.5 : JVM running times with memory hog.

6.3.5 Nicing-down memory hogs

As a final experiment, we niced-down the memory hog process by 14, just enough to push it into the next priority class (see Chapter 5). Then the JVM's heap grew to the original 220MB, while the memory hog replaced memory from its own footprint. Thus, the effects of this rogue process are isolated from the JVM.

6.4 Malloc-held unused memory

The malloc library in FreeBSD is designed with the ability to release completely free pages to the system. One of the data structures it maintains is a table of the count and bitmap of the chunks allocated on each virtual page. Pages with no chunks allocated on them can potentially be released to the system using `madvise(mem, MADV_FREE)`. However, without an arena-based allocator, this malloc library lacks the ability to coalesce free pages and return them to the system in bulk, so it needs to call `madvise` on pages one by one. Since this can be expensive in the general case, the default behavior is to *never* do it [39, 38].

We measured the extent to which common applications retain whole pages of unused memory. On our FreeBSD workstation, in our typical work environment with a slew of common applications running (X windows, mozilla, emacs, etc.), the average application is seen to hold *at least 5% of unused free pages* due to its malloc library not returning them to the system, and frequently 10% or more. Quantity-wise, the X windows server is the application with the most memory of this kind (up to 10

MB), while percentage-wise it is the window manager (almost always over 50% of its in-core memory).

It is not always possible to find free memory pages to release. Applications may have memory allocation and access patterns that result in small chunks allocated on a large number of virtual pages. This leads to internal fragmentation, which is discussed in greater detail in Section 5.3.2.

The FreeBSD malloc library allows the user to pass it a hint by setting the environment variable $MALLOC_OPTIONS=H at the time of executing a program. This instructs the library to invoke `madvise` immediately on pages that become completely free. This incurs the overhead of a system call for every page freed, and is therefore not optimal. We measured this overhead to be up to a factor of 8 in the worst case scenario. Postings on NetBSD forums have reported up to a factor of 4 degradation due to this effect [68], allegedly due to recent VM changes that are too expensive.

Our system attempts to achieve the best of both worlds: it enables the library to retain free memory in the general case, and release them when memory pressure is about to happen. We embed a SIGMEM handler into libc (without modifying applications or re-linking dynamically linked ones). On receiving SIGMEM notifications, this handler releases all the pages in the freelist by invoking madvise on contiguous sequences of free pages.

With these changes, dynamically linked applications on our desktop system collectively released 38MB of memory when they were notified of the potential onset of

memory pressure. These madvise invocations for contiguous extents of free memory happened in less than 0.5 seconds on the first notification, and in negligible time on subsequent ones. The released 38MB proced substantial on the 256MB machine, and resulted in some paging avoidance, and later, additional memory for the buffer cache.

6.5 Mozilla cache

This experiment demonstrates the applicability of memory adaptation to desktop environments. The Mozilla web browser maintains a memory cache of incoming web objects and pre-rendered images. We measure the loading time for a series of image-intensive web pages, starting from a cold cache and warm caches of small (10MB) and large (200MB) sizes. Using a small cache reduces the loading time by a factor of 3.5 due to the cached images, whereas a large cache reduces it by an additional factor of up to 2. For real web client workloads [54], benefits acquired through caching may vary depending on browsing patterns, and on whether the working set is closer to 10MB or 100MB. In general, when memory is plentiful, it is worthwhile using a large cache.

We implemented the memory resizing adaptation in Mozilla by partially taking the actions that "clear memory cache" does. Mozilla has some internal support for reacting to memory pressure: when its memory module detects allocation failures, or when it triggers a low memory condition (which it never does on Unix systems), then other subsystems become more conservative about memory usage. Since there

is no external interface connecting Mozilla to the system's memory management unit (such as a severity metric notification), this aspect of Mozilla is underdeveloped, and so we did not use it.

If the system first encounters memory pressure (severity drops from S_∞ to S_{max}), then a severity notification is received, and the browser shrinks its cache to a reasonable value of 30MB. Upon increasing memory pressure, the browser is able to release an additional 20MB of memory with modest performance degradation. It is actually cheaper to page these objects to and from disk than to fetch them over the network (milliseconds versus seconds); however, due to the low hit rates typically observed for web pages, our adaptation finds it cheaper to simply free a large fraction of the cache, bringing it down to 10MB as long as there isn't severe and sustained memory pressure. Therefore the application must factor in the cache hit rate while estimating the effective regeneration cost of its objects.

6.6 Evaluation of overhead in adaptive memory management

Our in-kernel design efficiently performs three key operations: (1) maintaining page ages, (2) selecting max-weighted-age page for replacement, and (3) nicing processes for memory. The only significant overhead is in scanning pages for page reference bits, some of which is continuously incurred rather than just in times of memory pressure. Our system scans pages only as frequently as their age itself, so the common-case overhead of scanning would be quite low. Indeed, on a system with 256MB RAM

running memory intensive applications such as a kernel build and a Mozilla web browser, the overhead of our system is as low as 0.1%.

When all the pages were very recently accessed, such as by a burst of activity, scanning takes the pathological overhead, which is observed to be a total 40ms in a scanning cycle of $S_{min} = 5s$, so the net overhead is 0.8%. This scales slightly super-linearly with memory, so on a 1GB machine, the equivalent overhead is measured to be a total 155ms, or 3.1%. In the interest of interactive applications, we modified it to scan in batches explicitly limited by a scanning time of 20ms, which causes an additional 0.3% and 1.1% overhead on 256MB and 1GB machines respectively.

6.7 Summary of evaluation

This chapter evaluates adaptive memory management over an instance of each of the following application types: caches, garbage collected virtual machines, malloc libraries, and interactive applications. In each case, we observed that the application stands to gain from liberally using as much memory as available. At the same time, significant performance improvements and system robustness gains were observed by releasing application memory when there is shortage, and tuning this memory release to correspond to the severity of memory pressure.

Chapter 7

Deceptive Idleness in Disk Schedulers

Disk schedulers have been an integral part of operating system functionality since the early days [20, 48, 52, 96, 119]. They perform the role of reordering disk requests in an attempt to improve performance, ensure fairness, etc., as per the goals of the scheduler. Disk drives have the property of being able to service requests to adjacent or nearby locations efficiently, but take significantly longer to seek between requests for distantly spaced on disk – this leads to the potential for seek optimization in a scheduler. Similarly, disk requests may arrive from multiple entities (such as processes), and the disk scheduler may wish to provide certain fairness or response time guarantees to each process about the service time or rate of its requests.

This chapter examines disk scheduling from a system-wide perspective, identifies a phenomenon called *deceptive idleness* and proposes *anticipatory scheduling* as an effective solution.

Disk schedulers are typically *work-conserving*, since they select a request for service as soon as (or before) the previous request has completed [97]. Now consider processes issuing disk requests *synchronously*: i.e. each process issues a new request shortly after its previous request has finished, and thus maintains at most one outstanding request at any time. This forces the scheduler into making a decision too early, so it assumes

that the process issuing the last request has momentarily no further disk requests, and selects a request from some other process. It thus suffers from a condition we call *deceptive idleness*, and becomes incapable of consecutively servicing more than one request from any process.

It is common for data requested by a process to be sequentially positioned on disk. Nevertheless, deceptive idleness forces a seek optimizing scheduler to multiplex between requests from different processes. The ensuing head seeks can cause performance degradation by up to a factor of four, as shown in the next section. In the related problem of proportional-share disk scheduling, meeting a given contract (i.e., a proportion assignment) may require the scheduler to consecutively service several requests from some process. Deceptive idleness precludes this requirement, thus limiting the scheduler's capacity to satisfy certain contracts. In both cases, the scheduler is reordering the *available* requests correctly, but system-wide goals are not met.

7.1 Problem description

This section describes and analyzes the phenomenon of deceptive idleness with two examples. In each case, the scheduler faces a shortage of the desired type of requests at critical moments.

Example #1: Seek reducing schedulers

Here is an example of how a seek reducing disk scheduler can degenerate to FCFS-like behaviour, and potentially suffer throughput loss by a factor of 4. Consider an operating system equipped with any seek reducing scheduler, like Shortest Positioning-Time First (SPTF) or CSCAN [119]. Let two disk-intensive processes p and q each issue several disk requests to separate sets of sequentially positioned 64 KB blocks. In the interest of seek reduction and throughput improvement, the scheduler would be expected to consecutively service many requests from p's set, then perform one expensive head repositioning operation, and service many of q's requests. This happens in practice, provided each process maintains one or more pending requests whenever the scheduler makes its decision: these moments in time are called *decision points*. Such an experiment results in a sustained throughput of 21 MB/s on our disk, owing to a service time of 3ms for every 64 KB block.

Now consider a scenario where the above requests are issued *synchronously* by the two processes, i.e., each process generates a new request a few hundred microseconds after its previous one finishes. A work-conserving disk scheduler never keeps the disk idle when there are any requests pending for service. It therefore tries to schedule some request immediately after (say p's) request has completed. At this decision point, process p has not yet been given the chance to perform the computation required to generate its next request. This forces the scheduler into choosing a request from q, performing a large head seek to that part of the disk, and servicing that request. The

subsequent request for p arrives soon after, but disk scheduling is non-preemptible, and it is now too late to service this nearby request. This leads to the scheduler alternating in an FCFS manner between requests from the two processes. Throughput falls to 5 MB/s, due to 9ms of average seek time and 3ms of read time for every 64 KB block. The problem persists even if more than two processes issue synchronous requests. In this case, CSCAN degenerates to a round-robin scheduler, whereas SPTF alternates between some pair of processes.

Example #2: Proportional-share schedulers

This example shows how deceptive idleness can affect schedulers in ways other than degrading throughput. Consider a proportional-share scheduler like Yet-another Fair Queueing (YFQ) [20], Stride Scheduling [115], or Lottery Scheduling [118]. Their intended behaviour is to deliver disk service to multiple applications (e.g., processes p and q) in accordance with an arbitrary preassigned ratio. For an assignment of 1:2 (or 33%:66%), the scheduler may service a few requests for process p, and correspondingly, about twice as many requests for process q. However, if these processes maintain only one outstanding request at critical moments, then as in the previous example, the work-conserving scheduler is forced to alternate between requests from the two processes. It becomes incapable of adhering to the desired contract for this workload, and instead achieves proportions much closer to 1:1. Figure 7.1 shows results of such an experiment. This effect on proportional-share schedulers has been noted

in [103] §5.6.

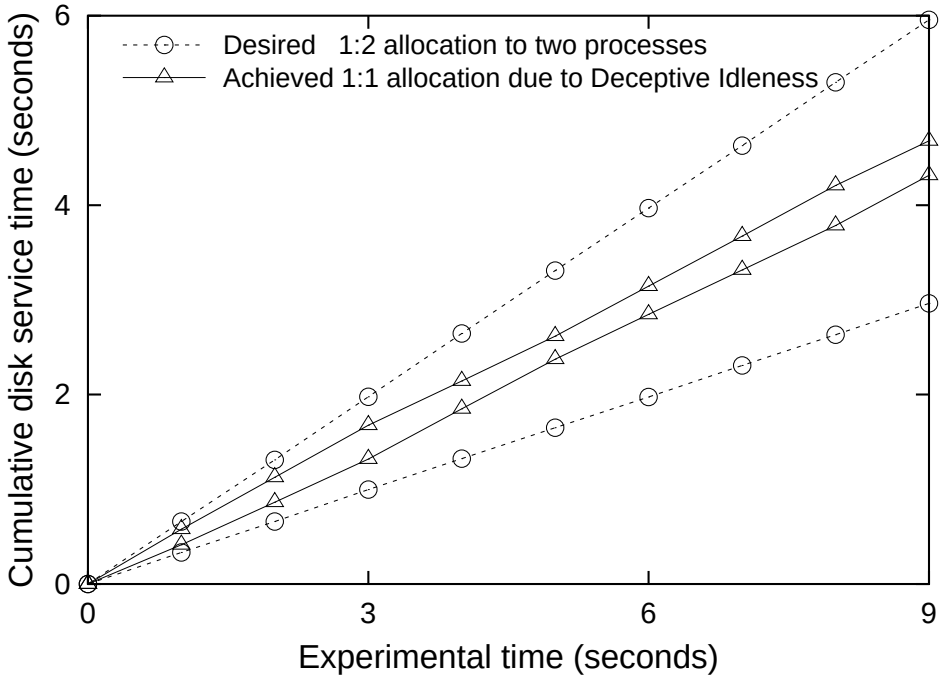


Figure 7.1 : A proportional-share scheduler: The outer pair of lines denote ideal scheduler response to an allocation ratio of 1:2 to the two processes. Inner pair of lines: synchronous I/O causes requests to be almost alternately serviced from the two processes, yielding proportions much closer to 1:1.

If we have three active processes instead, say p, q, r with shares of 1:1:3, then Stride will be forced to schedule requests from processes in the sequence r, p, r, q, etc. and achieve the skewed proportions of 1:1:2. For nontrivial reasons, a lottery disk scheduler under similar circumstances will deliver proportions of 2:2:3 instead (see [51] §4.2.2 for details).

7.1.1 The underlying problem

In both examples, the scheduler reorders the available requests according to its scheduling policy, but fails to meet overall objectives of performance and quality

of service. In essence, processes that issue synchronous requests cause the work-conserving disk scheduler to receive no requests from that process, in time for the following decision point.* This leads to deceptive idleness, rendering the scheduler incapable of exploiting spatial and temporal locality among synchronous requests.

7.2 Prefetching

It is possible to partially work around the problem of deceptive idleness by using *asynchronous prefetch*. This involves predicting the future request issue pattern for a process, and issuing its immediately forthcoming request before the current one completes. Each process thus maintains multiple outstanding requests at decision points, and gives the scheduler the chance to service consecutive requests from the same process. Seek reduction opportunities can therefore be exploited if requests issued by a process are sequential. Likewise, a proportional-share scheduler would now have the capacity to adhere to its contract, even for synchronous requests.

Prefetch can be effected either explicitly by the application or transparently by the kernel. However, both approaches have fundamental limitations in terms of feasibility, accuracy, and overhead.

*This happens despite the system-wide request queue generally being long and bursty on loaded server systems [96].

7.2.1 Application-driven prefetch

Applications can embrace programming paradigms and techniques that prevent the onset of deceptive idleness. They can use asynchronous I/O using APIs such as `aio_read()` to prefetch future requests. Alternatively, they can develop their own asynchronous I/O methods using multiple processes or kernel threads, to proactively issue disk requests of the right type (e.g., sequential).

There are several problems with this. (1) Applications are often fundamentally unaware of their future access pattern, and may be incapable of issuing accurate prefetch requests. Examples include filesystem metadata and database index traversals, and predicting future requests in webservers. (2) Applications may have to be written in a cumbersome programming paradigm, whereas most applications are better suited to a sequential programming style. (3) Existing applications would have to be rewritten for this purpose, which may not be desirable or even possible under some circumstances. (4) Issuing explicit read requests using Unix API functions (instead of memory mapping a file) may entail more data copying and cache pollution, which could become expensive for in-memory workloads. Lastly, (5) asynchronous I/O primitives may not be available in some operating systems.

7.2.2 Kernel-driven prefetch

Filesystems can (and most do) try to guess future request patterns for applications and issue separate asynchronous prefetch requests[†] for them. The usual reason is to overlap computation with I/O [99], but this prefetching also prevents deceptive idleness. There are, however, limitations to this transparent approach. The file system is typically even less capable of predicting future access patterns than applications are. Prefetch needs an exact notion of the location of the next request, and the penalties of misprediction can be high. This forces the prefetching to be complicated, yet conservative. Applications such as database systems can issue requests possessing spatial locality, but their access patterns may be extremely difficult to detect and effectively prefetch. Finally, sequentially accessed medium-sized files are often too small for the filesystem to detect sequential access and confidently issue prefetch requests. In summary, prefetching can potentially alleviate and even eliminate deceptive idleness, but limitations in its feasibility and effectiveness under many conditions discount it as a general solution.

Studies have shown an increasing trend in modern disk-intensive applications to issue non-sequential disk requests that nonetheless possess spatial locality [90, 114]. Prefetching has limited utility in these cases, and it is vital to consider complementary and more widely applicable alternatives.

[†]different from synchronous readahead, where requests are enlarged to 64 KB to amortize seek costs over larger reads.

Chapter 8

Anticipatory Disk Scheduling

This chapter proposes the *anticipatory disk scheduling framework*, and applies it to various disk scheduling policies. It solves deceptive idleness as follows: before choosing a request for service, it sometimes introduces a short, controlled delay period, during which the disk scheduler waits for additional requests to arrive from the process that issued the last serviced request. The disk is kept idle for short periods of time, but the benefits gained from being able to service multiple requests from the same process easily outweigh this loss in utilization. The framework is thus an application of the *non-work-conserving scheduling discipline*. The exact tradeoffs are sensitive to the original scheduling policy, so to determine whether and how long to wait each time, we propose adaptive heuristics based on a simple cost-benefit analysis.

We implement anticipatory scheduling as a kernel module in the FreeBSD operating system, evaluate it against a range of microbenchmarks and real workloads, and observe significant performance improvements and better adherence to quality of service objectives. For a trace-based, disk-intensive workload, the *Apache webserver* delivers between 29% and 71% more throughput by capitalizing on seek reduction within files. The synchronous, read-intensive phase of the *Andrew filesystem benchmark* runs faster by 54% due to seek reduction both between files and within each

file; consequently, the overall benchmark improves by 8%. Variants of the *TPC-B database benchmark* exhibit speedups between 2% and 60%: in the latter case, deviating from a standard TPC-B setup, by submitting read-only queries to multiple separate databases leads to more seek reduction opportunities. The *Stride* proportional disk scheduler [115] achieves its assigned allocations even for synchronous I/O (assuming there is sufficient load), and simultaneously delivers high throughput.

8.1 Key ideas

There are three necessary conditions for deceptive idleness to manifest itself: (a) multiple disk-intensive applications concurrently issuing synchronous disk requests, (b) the intrinsic non-preemptible nature of disk requests, and (c) a work-conserving disk scheduler, which schedules a request immediately upon completion of the previous request. Our solution takes the intuitive approach of eliminating condition (c), by wrapping a given disk scheduling policy in a non-work-conserving *anticipatory scheduling framework*.

When a request completes, the framework potentially waits briefly for additional requests to arrive, before dispatching a new request to the disk. Applications that quickly generate another request can do so before the scheduler takes its decision; deceptive idleness is thus avoided. The fact that the disk remains idle during this short period is not necessarily detrimental to performance. On the contrary, we will show how a careful application of this method consistently improves throughput and

adheres more closely to desired service allocations.

The question of whether and how long to wait at a given decision point is key to the effectiveness and performance of our system. In practice, the framework waits for the shortest period of time over which it expects, in high probability, for the benefits of waiting to outweigh the costs of keeping the disk idle. An assessment of these costs and benefits is only possible relative to a particular scheduling policy: a seek reducing scheduler may wish to wait for contiguous or proximal requests, whereas a proportional-share scheduler may prefer weighted fairness as its primary criterion. To allow for such flexibility, while minimizing the burden on the developer of a particular disk scheduler, the anticipatory scheduling framework consists of three components: (1) The original disk scheduler, which implements the scheduling policy and is unaware of anticipatory scheduling; (2) a scheduler-independent *anticipation core*; and, (3) adaptive scheduler-specific *anticipation heuristics* for seek reducing and proportional-share schedulers.

Figure 8.1 depicts the architecture of the framework. The anticipation core implements the generic logic and timing mechanisms for waiting, and relies on the anticipation heuristic to decide if and how long to wait. This heuristic is implemented separately for each scheduler, and has access to the internal state of the scheduler. To apply anticipatory scheduling to a new scheduling policy, one merely has to implement an appropriate anticipation heuristic.

The remainder of this section spells out our two assumptions about workload

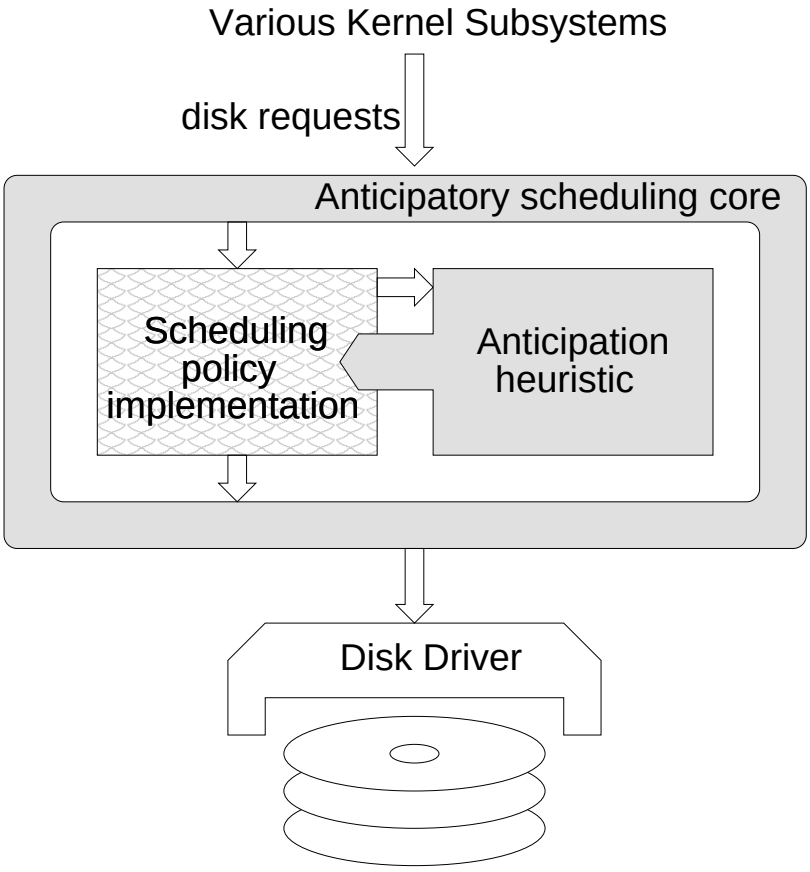


Figure 8.1 : Anticipatory scheduling framework

characteristics, then describes the anticipatory scheduling framework, followed by appropriate anticipation heuristics for seek reducing and proportional-share schedulers. Finally it covers some implementation issues.

8.2 Workload assumptions

We tentatively make two assumptions about the granularity at which applications issue related disk requests.

Assumption #1: *Synchronous disk requests are issued by individual processes.* In most applications, a dependence between disk requests is explicitly reflected in code structure, so it is uncommon for multiple processes to coordinate to issue a set of synchronous requests. This assumption serves two purposes: (a) it considerably simplifies the anticipation heuristic, by requiring it to wait only for the process whose request finished most recently, and (b) it allows the anticipation core to optimize for the common case when this process gets blocked upon issuing a synchronous request.

Assumption #2: *Barring occasional deviations, all requests issued by an individual process* have approximately similar degrees of spatial and temporal locality with respect to other requests from that process, and these properties do not change very rapidly with time.* The anticipation heuristic adaptively learns application characteristics on a per-process granularity; this assumption constitutes the basic requirement for adaptation to be possible. We therefore assume that a process does not interleave disk requests with markedly different locality properties.

Experimental results with real applications reported in Chapter 9 indirectly confirm that these assumptions hold to a sufficient degree. Relaxing these assumptions to accommodate a larger range of workloads and access patterns is the subject of future work. Some ideas in this direction are proposed in Section 9.9.2.

---

* *One may substitute* process *for* thread *here, similar to what is proposed in Section 9.9.2, if the system wishes to support reasonable anticipation of requests from multithreaded applications where each thread may have a different disk access pattern.*

8.3 The anticipation core

A traditional work-conserving scheduler has two states, IDLE and BUSY, with transitions on scheduling and completion of a request. Applications can issue requests at any time; these are placed into the scheduler's pool of requests. If the disk is idle at this moment, or whenever another request completes, a request is *scheduled*: the scheduler's *select* function is called, whereupon a request is chosen from the pool and dispatched to the disk driver.

The anticipation core forms a wrapper around this traditional scheduler. Whenever the disk becomes idle, it invokes the scheduler to select a candidate request (as before). However, instead of dequeuing and dispatching immediately, it first passes this request to the anticipation heuristic for evaluation. A result of zero indicates that the heuristic has deemed it pointless to wait; the core therefore proceeds to dispatch the candidate request. However, a positive integer represents the waiting period in microseconds that the heuristic deems suitable. The core initiates a timeout for that period, and enters the new WAIT state. Though the disk is inactive, this state differs from IDLE by having pending requests and an active timeout.

If the timeout expires before the arrival of any new request, then the previously chosen request is dispatched without further ado. However, new requests may arrive during the waiting period; these requests are added to the pool. The anticipation core then immediately asks the scheduler to select a new candidate request from the pool, and asks the heuristic to evaluate this candidate. This may lead to immediate

dispatch of the new candidate request, or it may cause the core to remain in the WAIT state, depending on the scheduler's selection and the anticipation heuristic's evaluation. In the latter case, the original timeout remains in effect, thus preventing unbounded waiting by repeatedly retriggering the timeout. The state diagram in Figure 8.2 illustrates this decision process.

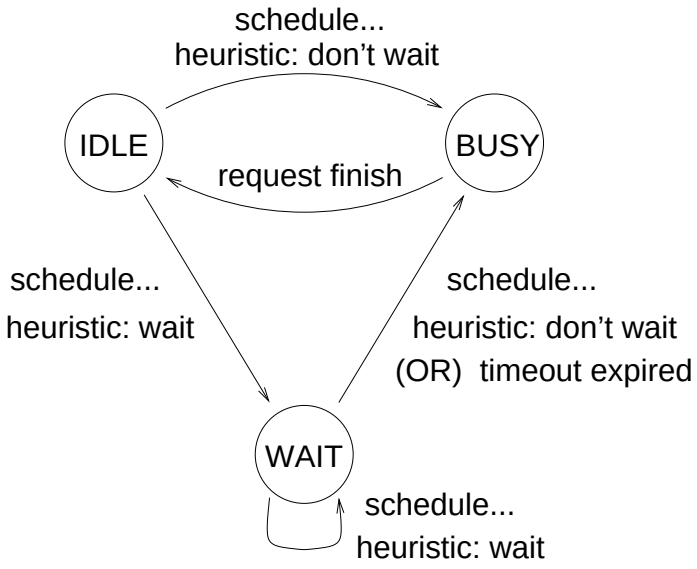


Figure 8.2 : Waiting mechanism, state diagram

There is a scheduler-independent optimization of the above algorithm: if the process whose request finished most recently blocks on I/O by issuing a synchronous request, then assumption #1 suggests that a dependent request will not arrive from any other process. The anticipation heuristic can thus be short-circuited, and the chosen request immediately dispatched. This happens quite often in practice, even on occasions when the heuristic would have decided to wait further.

8.4 Seek reducing schedulers

This section describes scheduler-specific anticipation heuristics for seek reducing schedulers such as SPTF, Aged-SPTF and CSCAN. The Shortest Positioning-Time First policy [119] (also known as Shortest Time First [96] and Shortest Access-Time First [52]) calculates the positioning time for each available request from the current head position, and chooses the one with the minimum. Our goal is to design an anticipation heuristic that maximizes the expected throughput.

The heuristic needs to evaluate the candidate request chosen by the scheduling policy. The intuition is as follows: if this candidate request is located close to the current head position, then there is little point in waiting for additional requests. Otherwise, using assumption #1, if the process whose request completed most recently is likely to issue the next request soon (i.e., its expected median thinktime[†] is small), *and* if that request is expected to be close to the current head position, then the heuristic decides to wait for it. The waiting period is chosen as the expected 95-percentile thinktime, within which there is a 95% probability that a request will arrive.

This simple idea is generalized into a succinct cost-benefit equation, intended to handle the entire range of values for positioning times and thinktimes. Our throughput objective translates to profitably balancing the benefit of waiting, i.e., expected gains in positioning time, against the cost of waiting, which is the additional time

[†]We define thinktime for a process issuing a request as the interval between completion of the previous request issued by the process and issue of a new request.

likely to be wasted. If LP (last process) is the process whose request has finished most recently, and *elapsed* is the time passed since completion of the previous request, then:

$$
\begin{aligned}
&benefit = (calculate\_positioning\_time(Candidate) \\
&\qquad - \text{LP}.expected\_positioning\_time) \\
&\\
&cost = \max(0, \\
&\qquad \text{LP}.expected\_median\_thinktime - elapsed) \\
&\\
&waiting\_duration = \max(0, \\
&\qquad \text{LP}.expected\_95percentile\_thinktime - elapsed) \\
&\\
&return\ (benefit > cost\ ?\ waiting\_duration\ :\ 0)
\end{aligned}
$$

Positioning time for the candidate request is calculated with a suitable estimator (more on this in Section 8.7). Regarding the cost estimate: for requests that arrive before the median thinktime, the heuristic expects progressively shorter periods of additional waiting; hence *elapsed* is subtracted from the expected median thinktime. However, if we wait beyond this median, the heuristic expects a request to be issued sometime very soon, and cost at this point becomes zero. Secondly, in the WAIT state, the anticipation core prevents unbounded waiting by not retriggering the timeout according to the heuristic's evaluation. Yet we calculate the correct value of waiting_duration: this is done to allow for coarse-granularity timers, so that a request arriving after the 95%ile thinktime will force an immediate dispatch as if the timeout had occurred just then.

The adaptive component of the heuristic consists of collecting online statistics on

all disk read requests, to estimate the three expected times. The expected positioning time for each process is a weighted average over time of the positioning time for requests from that process, as measured upon request completion. The decay factor is set to forget 95% of the old positioning time value after ten requests, so the heuristic adapts fast. An alternate, approximate method is to track the expected *seek distance* of a request from the previous request issued by that process, and calculate expected positioning time on the fly.

Expected median and 95%ile thinktimes are estimated by maintaining a decayed frequency table of request thinktimes for each process. Thinktimes are computed from the time of completion of the last request issued by a given process, to the current time. If, however, the scheduler already has a read request queued for this same process, then this new request is treated as asynchronous and its thinktime is set to zero. The heuristic maintains 30 per-process buckets that store the count of requests that arrive after various thinktimes, ranging from 0 to 15ms at a granularity of 500μs per bucket. These bucket counts are all decayed by reducing them to 90% of their original values for every incoming request for that process. The distribution of thinktimes usually looks like a bell curve; this is consistent with assumption #2. (For many applications, the crest is located at about 1ms). The heuristic calculates the median and 95%ile points of this curve; it does all the above for every incoming synchronous request.

This heuristic is suitable for the conceptually simple SPTF policy. We now con-

sider modifying it for two other seek reducing schedulers, namely Aged-SPTF and CSCAN.

8.4.1 Variant: Aged-SPTF

SPTF is known to suffer from potential starvation, since requests for distant locations on the disk may never get serviced. To bound response time, Aged-SPTF (also known as Aged-SATF and Weighted-SPTF) has been proposed as a variant: requests in the SPTF queue are associated with priorities, which are raised in some manner (often gradually) with queued time. A request with sufficiently high priority overrules the SPTF decision and gets scheduled [52, 96, 119].

The anticipation heuristic for SPTF works for Aged-SPTF also, with one minor limitation. When Aged-SPTF chooses a distant request that is too old, the SPTF heuristic would be unaware of this. It may decide to wait for additional, nearby requests. However, even if a new request from the last process arrives in this period, the scheduler then continues to pick the same old request. The last process then gets blocked, and the scheduler-chosen candidate is serviced as desired. This incurs one unnecessary thinktime on each of such occasions; this minor performance problem can be fixed by customizing the heuristic to the Aged-SPTF policy: whenever Aged-SPTF selects a request that is different from the request that SPTF would correspondingly choose, then we decide not to wait.

8.4.2 Variant: CSCAN: Cyclic SCAN

CSCAN (also known as C-LOOK) is an extremely popular scheduling policy, and is implemented in many Unix-based operating systems. It is the unidirectional version of Elevator/SCAN/LOOK; it moves the head in one direction, servicing all requests in its path, and then starts over at the first available request.

Our anticipation heuristic for this scheduler is based on the one for SPTF, with one additional clause. The statistics collection module in the heuristic additionally maintains a decayed expectation of the seek direction: forward or backward. On evaluating a request, if the current candidate involves a forward seek and the expected next request has a fairly high likelihood (more than 80%) of a backward seek, then we bypass the cost-benefit equation and decide not to wait. In the opposite case, we wait for the usual amount of time. For applications performing random access, with roughly 50% of the seeks pointing in each direction, this heuristic for CSCAN is not ideal. This is because CSCAN itself is poorly suited to handle this case.

8.5 Proportional-share schedulers

We next present an anticipation heuristic designed for a proportional-share scheduler like Yet-another Fair Queueing (YFQ) [20] or Stride [115]. These policies maintain weighted virtual clocks to remember the amount of disk service received by each process. A request is chosen from the process with the smallest virtual clock, so as to advance them in tandem.

Unfortunately, deceptive idleness forces these virtual clocks to go out of sync. Some processes do not generate enough requests in time, and their virtual clock lags behind. Processes that genuinely issue few disk requests also lag behind, but their expected thinktimes are high. Our heuristic is therefore as simple as waiting for the last process, if it meets three conditions: (a) it has no pending requests at the time its last requests completes, (b) it has an expected thinktime smaller than 3ms, and (c) it has a virtual clock smaller than the minimum virtual clock of processes with available requests (*minclock*). The 3ms threshold is chosen somewhat arbitrarily; there is no consistent way to balance weighted fairness against performance. 3ms is observed to be larger than the thinktimes for most applications, without being too large as to degrade performance. As before, we wait for the 95%ile point of the thinktime distribution for this process.

8.6 Heuristic combination

A proportional-share scheduler with an assignment of 1:2 can service one request from the first process for every two requests from the second. Alternatively, it can enable some seek reduction, by slightly relaxing the timescale on which it operates. This allows the scheduler to service n requests for the first process for every $2n$ requests for the second, where each set might contain sequential requests. One variation on this theme is suggested in [113], where the scheduler picks from processes with virtual clocks between *minclock* and $minclock + \tau$ (where τ is a relaxation threshold, and

could be 1 second). Among these, it chooses the request with the smallest positioning time.

We propose a *combination heuristic* for this scheduler, thus hinting at general methods of combining anticipation heuristics. This combination is necessary because: (1) the heuristic for SPTF, if applied directly here, would not wait for either process if the access pattern were random, and would thus violate the proportion assignment; (2) the heuristic for Stride, if used, would wait only for the process with higher share, and thus enable only partial seek reduction.

A straightforward approach of combining these two heuristics is to separately evaluate the candidate request on each of the two, and return the larger of the two evaluations. In other words, if the waiting decision is taken for either reason, then the combination will conservatively choose to wait.

We identify and accommodate for two minor performance issues with this simplistic approach. Firstly, the Stride policy has been relaxed due to the introduction of τ. Condition (c) in the anticipation heuristic for Stride needs to be correspondingly changed from $minclock$ to $minclock + \tau$.

Secondly, consider the heuristic for SPTF waiting for sequential requests from process p, and successively servicing many such requests. At some point, p's virtual clock may become larger than $minclock + \tau$, in which case the conservative decision to wait becomes pointless. Our heuristic watches for this condition and decides not to wait.

8.7 Implementation issues

There are two implementation issues that deserve elaboration, namely calculating positioning time for requests and building an inexpensive timeout mechanism.

Estimating access time for requests is nontrivial due to factors like rotational latency, track and cylinder skews, and features of modern disks like block remapping and recalibration. Nonetheless, much work has been done in this area, and it is possible to build a software-only predictor with over 90% accuracy [48, 52, 94, 120]. However, we used a much simpler logical block number based approximation to positioning time. A user-level program performs some measurements to capture the mapping between the logical block number difference between two requests and the corresponding head positioning time, and fits a smooth curve through these points. This takes about 3 minutes at disk installation time, but can be made online and non-intrusive. This method automatically accounts for seek time, average rotational latency and track buffers. It has an accuracy of about 75%, which we experimentally confirm to be sufficient, given the insensitivity of the anticipation heuristic.

There are many possible timer mechanisms to choose from. We use the i8254 Programmable Interval Timer (PIT) to generate interrupts every 500μs, and build a simple timeout system over that. Experiments demonstrate how this rather coarse-grained timer is amply sufficient for our purposes. Each interrupt causes a processing overhead of about 4μs on our hardware [7], thus causing about 1% CPU overhead on computational workloads. Other timeout mechanisms can be used in place of

the i8254, if higher accuracy and lower overhead are desired. Some pentium-class processors (mostly SMPs) have an on-chip APIC that delivers fine-grained interrupts with an overhead of only 1 to 2μs per interrupt. Alternatively, soft-timers [7] pose an extremely light-weight alternative.

Chapter 9

Evaluation of Anticipatory Disk Scheduling

This chapter evaluates the anticipatory scheduling framework on a range of microbenchmarks and real workloads. We show that this transparent kernel-level solution eliminates deceptive idleness, and achieves significant performance improvement and closer adherence to QoS objectives wherever applicable.

Code and platform: We implemented the anticipatory scheduling framework and heuristics in the FreeBSD-4.3 kernel. The code comprises of a kernel module of about 1500 lines of C code, and a small patch to the kernel for necessary hooks into the scheduler and disk driver. Unless otherwise stated, our experiments are conducted on a single 550MHz Pentium-III system, equipped with a 7200rpm IBM Deskstar 34GXP IDE disk and 128 MB of main memory.

Schedulers: All experiments with a seek reducing scheduler use Aged-SPTF unless otherwise specified. We configure this scheduler to perform shortest positioning-time first scheduling, with a bounded per-request latency of 1 second. This is found to achieve performance to within 1% of SPTF. Anticipatory scheduling involves an intrinsic latency tradeoff: servicing multiple requests from one process for seek reduction necessarily increases request turnaround time for another. However, most server-type applications would find this small increase acceptable, in exchange for

significant improvements in throughput. A system that desires lower latency may reduce the above delay bound to say 100ms; this was measured to reduce the throughput by at most 8% on our system.

Metrics: Our experiments employ two metrics of application performance: the application-observed throughput in MB/s, and the disk utilization. In our framework, a disk spends time either servicing requests (i.e., positioning head and transferring data) or idling; we define *disk utilization* in an interval as the percentage of real time spent servicing requests.* This choice of the utilization metric depicts the fraction of time that the disk is deliberately kept idle, and helps in understanding some throughput measurements.

Turning off filesystem prefetch: Some operating systems, including FreeBSD, do not implement asynchronous prefetch in some subsystems. For example, the VM subsystem does not issue auxiliary prefetch requests for page faults that are serviced from disk. Similarly, FreeBSD also does not perform asynchronous prefetch for `sendfile()` and `readdir()`. This allows us to effectively turn off prefetching for evaluation purposes, by mapping files to memory and accessing the memory locations.

Two sets of microbenchmarks, exhibiting variations in access patterns and think-times, serve to illuminate the workings of anticipatory scheduling as applied to seek reducing schedulers.

*In contrast, a work-conserving scheduler never idles for a busy workload, and might prefer to define utilization as the percentage of service time spent transferring data from disk.

9.1 Microbenchmark: Access patterns

We study the effect of anticipatory scheduling on synchronous requests issued in different access patterns, with and without filesystem prefetch enabled. Two processes rapidly issue 64 KB disk read requests into separate 1 GB files; these are either sequential (*seq*), or target every alternate 64 KB chunk (*alter*), or are randomly positioned within their respective files (*random*). Some experiments use the `read` system call, for which FreeBSD 4.3 transparently issues asynchronous prefetch requests if the access pattern is detected to be sequential on disk. Other experiments map their file into memory using `mmap`, and fault on thir corresponding memory pages; these are not subject to asynchronous prefetch. Figure 9.1 shows the results.

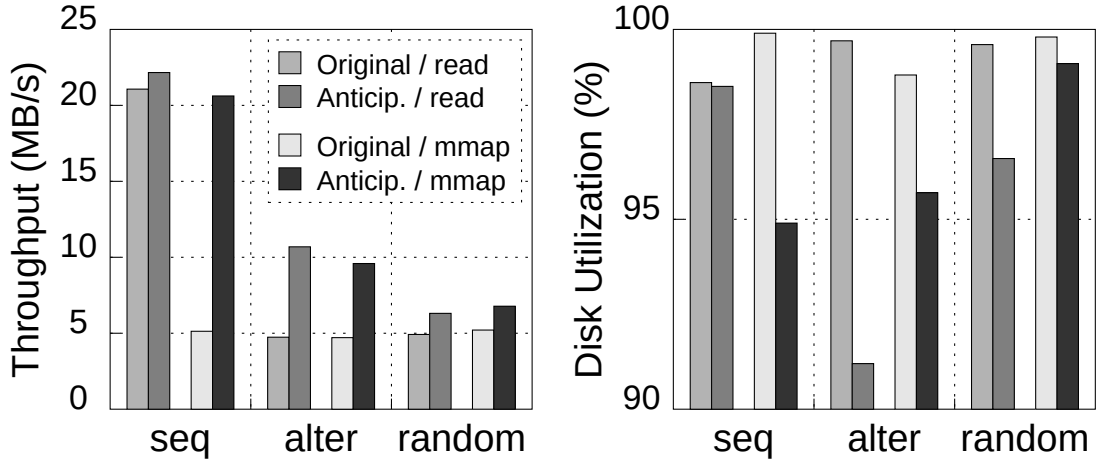


Figure 9.1 : Impact of anticipatory scheduling on disk throughput and utilization, using sequential, alternate-block and random access workloads, and read versus mmap based access.

Asynchronous prefetch ensures that sequential accesses (*seq* in the figure) using read achieve almost full disk bandwidth (about 21 MB/s). However, filesystems often

lay out logically contiguous blocks of a large file as a set of separate regions on disk. On the infrequent occasions that a boundary is crossed, FreeBSD's prefetching mechanism temporarily assumes non-sequential access and conservatively backs off. Anticipatory scheduling waits for such processes, thus exploiting spatial locality within the large file. Performance improves by about 5%, by steadily fetching blocks from one file until Aged-SPTF forces it to switch.

Since memory mapped accesses in FreeBSD are not subject to asynchronous prefetch, anticipatory scheduling attains four times better throughput than the original case. This achieves throughput almost equal to the maximum disk bandwidth; the 6% difference between the two is reflected by an almost equal fraction of time that the disk is kept idle. This lack of asynchronous prefetching `mmap` case is arguably a shortcoming of FreeBSD's prefetch implementation, which may be fixed in the future. However, as exemplified in the following two cases of *alter* and *random*, non-sequential disk access using `read` can use anticipatory scheduling to significantly improve throughput wherever prefetching fails.

Consider the second set of experiments (*alter* in the figure), where alternate blocks are read. This defeats the FreeBSD prefetch heuristic, causing both read and mmap to achieve only 5 MB/s. Anticipatory scheduling improves throughput to the maximum that can be achieved for alternate blocks, i.e., half the disk bandwidth. We will see several variants of such non-sequential access in real workloads.

Lastly, in the random access case (*random* in the figure), the smaller improvements

(28% and 30%) by anticipatory scheduling are because each process is performing random access within its respective file, so gains are mostly due to seek reduction between files.

9.2 Microbenchmark: Varying thinktimes

The next set of four microbenchmarks illustrates the impact of waiting on applications that take different amounts of time to issue the next request. Two processes map separate, large files into memory, and fault on these memory pages sequentially (thus without asynchronous prefetch). After every 64 KB, they pause for some amount of time as described below.

9.2.1 Symmetric processes

Consider Figure 9.2, where time t on the horizontal axis represents the duration in milliseconds that *each* process spends waiting between requests. Each data point in the throughput graph is a separate experiment. For values of t up to 8ms, the original system alternates between requests from the two processes, achieving only 5 MB/s. When thinktime exceeds 8ms, the waiting time becomes comparable to request service time, and utilization for the original system starts falling below 100%. Occasionally, deceptive idleness is avoided by servicing two successive requests for the same process. This fades away for larger values of t, as requests in the queue are issued by one of several processes.

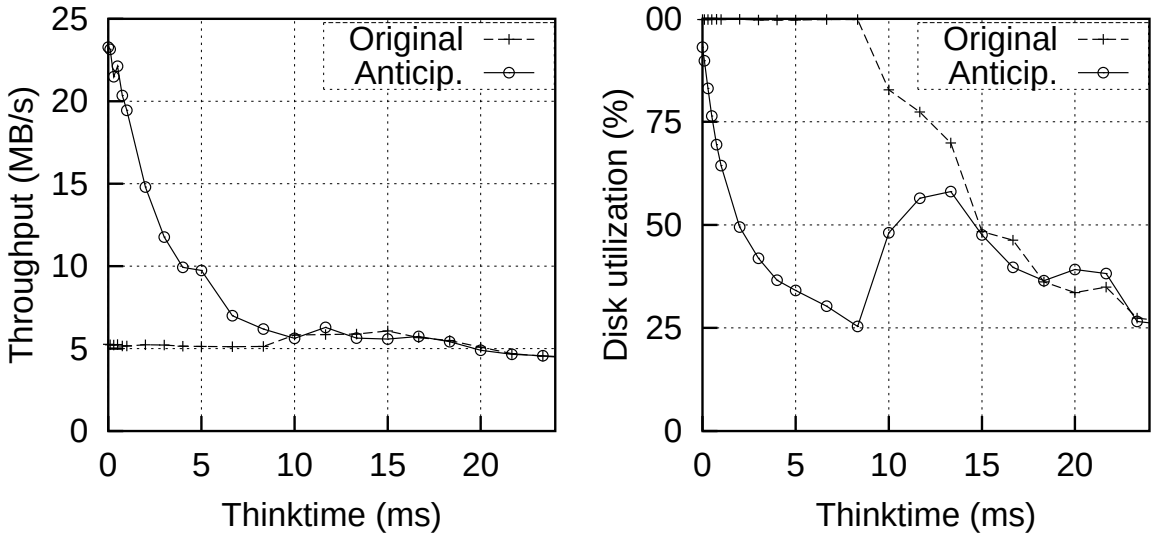


Figure 9.2 : Increasing thinktimes for both processes

With anticipatory scheduling enabled, the situation changes as follows: When $t = 0$, we see the familiar situation where throughput is four times that of the original system. For larger values of t up to 8ms the effect of waiting becomes increasingly burdensome on throughput and utilization, and the improvement steadily declines. At about 8ms, the waiting time becomes comparable to request service time, and the cost-benefit equation tips the other way. Performance then approaches that of the original system to an increasing degree. Measurements indicate that many applications have very short thinktimes when busy, in the region of 200μs to 2ms. Hence, anticipatory scheduling is expected to achieve significant benefits on real applications.

9.2.2 Asymmetric processes

Consider an alternative scenario in Figure 9.3 where only one (slow) process waits for duration t between requests, while the other (quick) process issues a request as soon as its previous request completes. The original system alternates between the two processes' requests for t up to 12ms, but beyond that, two or more requests arrive from the quick process for every request from the slow one. This causes partial avoidance of deceptive idleness, due to which performance gradually improves for increasing t.

With anticipatory scheduling enabled, the attained throughput exceeds that of the original system by a large margin. The anticipation heuristic is greedy, and for small values of thinktime, it decides to wait for both processes. This results in a gradual throughput decrease with increasing thinktime, until a point is reached (4ms) where the heuristic waits for the quick process but not for the slow process. Throughput rises back to the maximum, with requests from the slow process serviced only when Aged-SPTF induces a switch. Note that Aged-SPTF only guarantees non-starvation, not fine-grained fairness.

9.2.3 Random thinktimes

Next, we seek to understand how well the anticipation heuristic adapts to thinktimes that vary rapidly within an experiment. Interestingly, if a process waits for a random duration uniformly distributed between 0 and t, it performs almost as well as the case when a process always waits for exactly the same amount of time. This is because

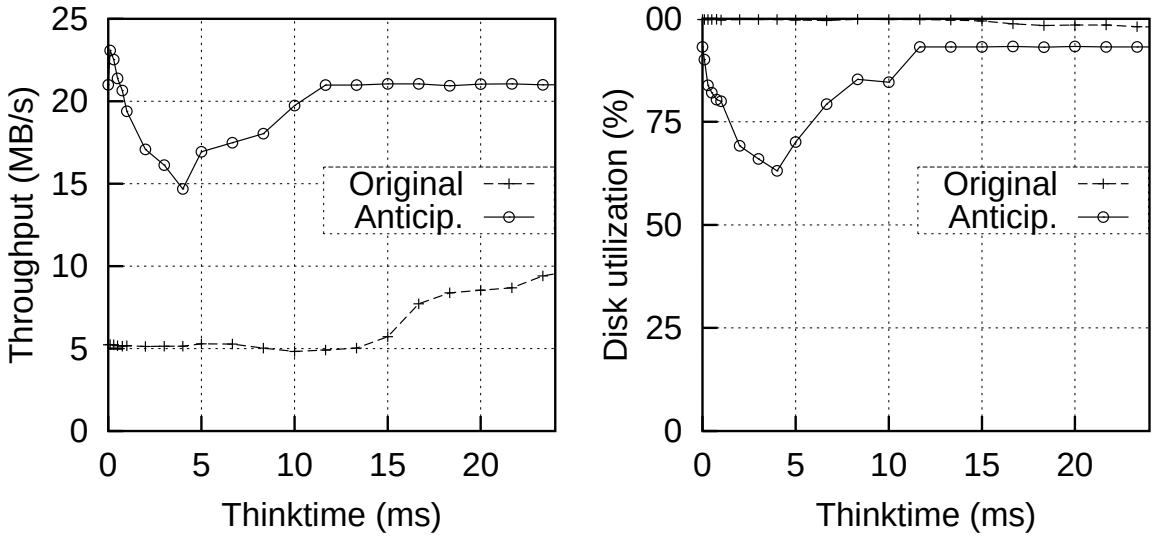


Figure 9.3 : Increasing thinktimes for one process

the expected median thinktime is judged to be roughly $t/2$, and the expected 95%ile thinktime becomes almost t.

9.2.4 Adversary

Since the heuristic copes with randomly varying thinktimes, we try to exercise the pathological-case behaviour of the heuristic by writing an intelligent adversary. Two symmetric processes wait for a duration determined as follows: they issue n rapid requests, then wait for a duration that just exceeds the timeout set by the heuristic, and repeat. This application actively fails to comply with assumption #2, and thus encumbers the heuristic from adapting effectively. Results for varying n are shown in Figure 9.4. For $n = 0$, the anticipatory scheduler can cope with all requests arriving slowly. But for n between 1 and 4, the anticipation heuristic performs only slightly

worse than the original system: by about 20%. This result indicates that even for pathological cases, or when the assumptions in Section 8.2 do not hold, the possible performance degradation is acceptably small.

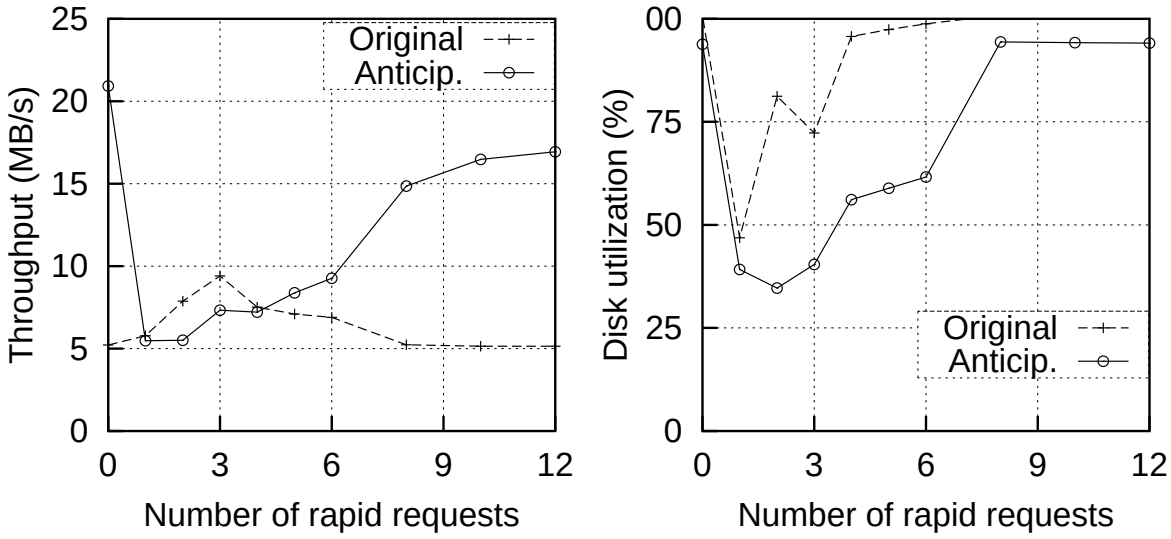


Figure 9.4 : Adversary application

The adversary issues several requests rapidly, followed by a long wait. Interestingly, a similar situation arises in practice when applications issue very large read requests (say 1 MB), and the FreeBSD kernel breaks them up into 128 KB chunks. In this case, the scheduler receives eight 128 KB requests in rapid succession, followed by the application's typically larger thinktime period. We solve this special case by having the filesystem flag such requests, whereupon the anticipatory scheduling core treats them like one large request.

The adversary application causes many timeouts to expire, and thus stresses the accuracy of the timer. In order to understand the sensitivity of our results to the

timer frequency, we reran the experiment with timer granularities of 50μs, 200μs, 500μs, and 1ms. Although the throughput peaked at 500μs (because larger timeouts allow for the occasional heuristic error), the greatest difference we saw among the four trials was only 10%. This was also supported by a similar experiment with the Apache webserver, where the difference was negligible.

Solving deceptive idleness can clearly bring about significant benefits on microbenchmarks, but what is its impact on real applications? To see this, we use two real applications (webserver and linker), and two standard benchmarks (filesystem and database) that we expect to reflect a wide range of application workloads.

9.3 The Andrew filesystem benchmark

The Andrew Benchmark [47] attempts to capture a typical fileserver workload in a software development environment. It consists of k clients, each performing five phases: (a) *mkdir*, which creates n directories, (b) *cp*, which copies a standard set of 71 C source files to each of these n directories, (c) *stat*, which aggressively lists all directory contents, (d) *scan*, which reads all these files using `grep` and `wc`, and finally (e) *gcc*, which compiles and links them. We configured n to be 500, so that the repository size exceeds main memory. We call this set of n directories a *repository*, and instantiate one such repository for each of the $k = 2$ clients, aiming to simulate concurrent access to a fileserver. This experiment uses the same Aged-SPTF scheduler as before, with and without anticipatory scheduling enabled.

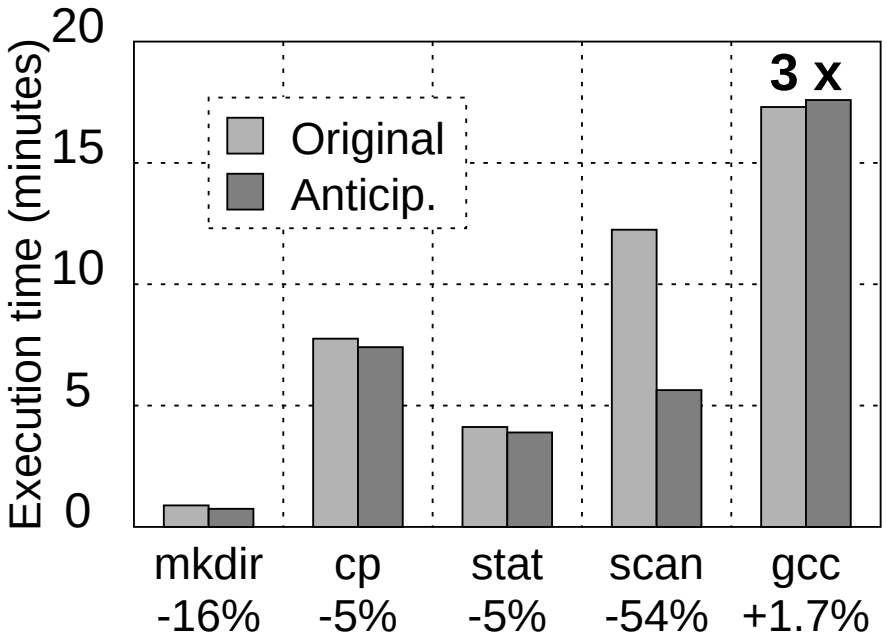


Figure 9.5 : The Andrew Benchmark. The last pair of bars are shown scaled down by a factor of 3.

A breakup of the execution times for individual benchmark phases is presented in Figure 9.5. Consider the scan phase, which is the only one that issues streams of synchronous read requests. Anticipatory scheduling transparently reduces execution time for this phase by 54%. Both `grep` and `wc` on FreeBSD use `read`, not `mmap`, and would thus benefit from kernel prefetch. However, individual files are small, so this prefetch has little effect. Major seek reduction happens here due to the files being in the same directory, and thus closely positioned on disk. Anticipatory scheduling enables the scheduler to capitalize on these seek opportunities and halve the execution time.

Other disk-intensive phases improve by smaller amounts: 16% for mkdir with metadata writes, and 5% for cp and stat each (the latter typically gets cached in

memory). The gcc phase is CPU-bound, but also performs some disk I/O; this aptly demonstrates the overhead of our system. There is an increase in execution time by 1.7%, due to two factors: CPU processing for the additional i8254 timer interrupts, and the CPU overhead corresponding to the heuristic execution routines (mainly statistics collection). This phase strongly dominates total execution time, but the overall benchmark still shows an improvement of 8.4%.

Performance with one client is the same with or without anticipatory scheduling; indeed, when there is only one stream of synchronous requests, anticipatory scheduling plays no role. Increasing the number of clients from 2 to 8 shows almost no performance difference: the scan phase improves by 57% in the latter case. This confirms the applicability and scalability of anticipatory scheduling to busy fileservers.

9.4 The Apache webserver

The Apache webserver employs a multi-process architecture to service requests from clients. Requests that miss in the main-memory cache are serviced from disk by the respective process. This happens frequently for webservers with large working sets, to the point of becoming disk-bound. In its default configuration, Apache-1.3.12 (and also 2.0a9) mmaps files that are smaller than 4 MB, and writes them out to a network socket. For larger files, Apache reads the data into application buffers first; this was done to prevent a swap-based DoS attack on IRIX systems. Many other webservers and ftp servers use similar mechanisms for file transfer.

We first configure Apache to exclusively use either read or mmap in a given experiment. We run the Apache webserver with 3 client machines which host 16 client processes each. Real websites have different amounts of concurrency, depending on amount and characteristics of incident load; we therefore varied the number of clients over a wide range, and observed very little difference in results. These clients rapidly play requests from a trace selected from the CS department webserver at the University of California, Berkeley [15]. These requests have a median size of 4768 bytes, a mean size of 86 KB, and a mean size of 13 KB if the largest 5% of the requests are excluded. This trace is quite disk-intensive, so 1000 requests target 745 distinct files. The scheduler, as before, is Aged-SPTF.

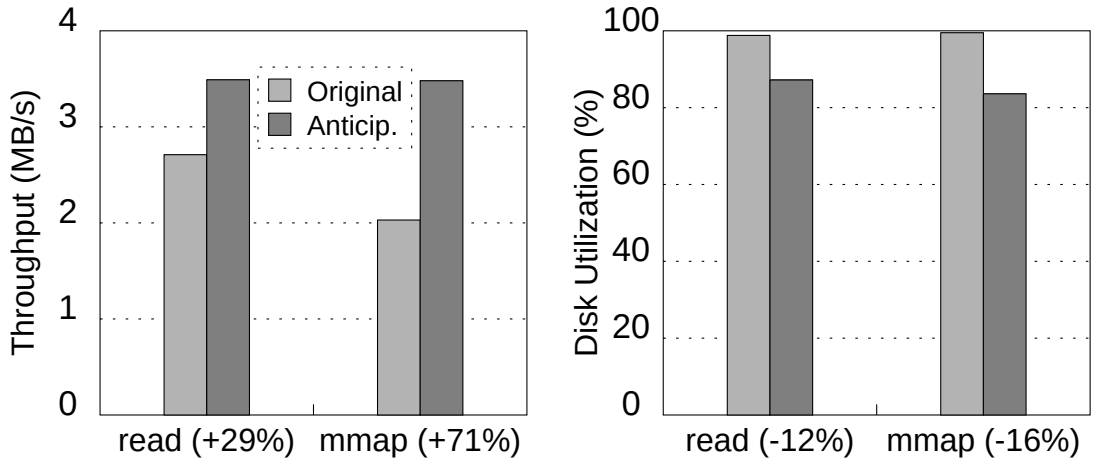


Figure 9.6 : The Apache Webserver configured in two modes, read and mmap. The former exemplifies the practical limitations of filesystem prefetch.

Figure 9.6 characterizes the observed throughputs and utilizations. We observe a 29% improvement in throughput for read, where anticipatory scheduling complements filesystem prefetch, and a larger 71% improvement for mmap (without prefetch). Un-

like in the Andrew Benchmark, all Apache clients generate requests to the same repository, so requests to an individual Apache process do not exhibit much locality *across files*. So seek reduction opportunities are mainly in terms of servicing each file fully before moving on to the next. Many files are too small for any seek reduction. *Intermediate-sized files* are potential candidates for prefetching, but filesystem prefetch is conservative and does not occur until a threshold number of requests are found to be sequential. Anticipatory scheduling effects the 29% improvement in this domain. Prefetch occurs for reads on large files, but not for mmap. This accounts for the large difference in performance between the two methods of access. In the default configuration (with mmap or read depending on file size), Apache yields 2.2 MB/s on the original system and 3.5 MB/s with anticipatory scheduling; this improvement of 59% lies between those for the read and mmap cases.

9.5 The GnuLD linker

The next experiment involves the last stage of a FreeBSD kernel build, starting from a cold filesystem cache. The GNU linker reads 385 object files from disk. 75% of these files are under 10 KB, whereas 96% are under 25 KB. After reading all their ELF headers, GnuLD performs up to 9 (but usually about 6) small, non-sequential reads in each file, corresponding to each ELF section. These reads are separated by computation required for the linking process.

The experiment in Figure 9.7 demonstrates the performance of one and two si-

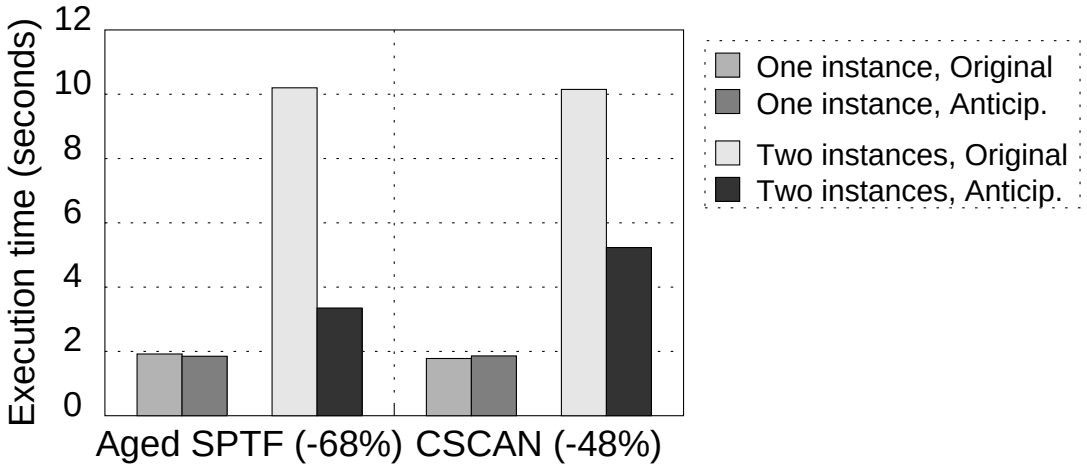


Figure 9.7 : The GNU Linker: multiple, concurrent instances cause deceptive idleness, which is eliminated by anticipatory scheduling.

multaneous instances of GnuLD on disjoint repositories. We use two schedulers this time, Aged-SPTF and CSCAN, to demonstrate the impact of their respective heuristics. With one synchronous request issuer process, both schedulers result in execution times of about 1.8 seconds each. We would normally expect this to double for two instances of GnuLD. However, deceptive idleness causes an increase in execution time by a factor of 5.5 instead. This is again because non-sequential accesses preclude transparent filesystem prefetching.

Anticipatory scheduling brings about a benefit of 68% in the Aged-SPTF case, and causes performance to scale almost exactly as expected (i.e., to twice the execution time of a single process). The CSCAN scheduler, on the other hand, always services requests in the forward direction. But object files are accessed in arbitrary order; CSCAN therefore intrinsically precludes anticipatory scheduling from attaining the

full potential for seek reduction. We see a performance improvement of 48%; this execution time is 56% higher than the Aged-SPTF case.

9.6 The TPC-B database benchmark

The TPC-B benchmark, specified by the Transaction Processing Council in 1994, exercises a database system on simple, random, update-intensive operations into a large database, and is intended to reflect typical bank transactions [29]. Though it is considered outdated, it serves to illustrate the impact of anticipatory scheduling on a read-write workload.

We implement the above with a MySQL database and two client processes. However, we somewhat deviate from the setup specified in TPC-B; our main goal is to demonstrate the gains due to anticipatory scheduling, rather than to obtain performance data for our hardware configuration. (1) Individual records in the database are required to be *at least* 100 bytes large. MySQL has computational overheads that made it CPU-bound for record sizes of 100 bytes, so we use 4 KB records to make data I/O the bottleneck. (2) We use a database size of 780 MB, thus considerably exceeding the 128 MB main memory size; our hardware is capable of supporting larger databases. (3) MySQL does not support transactions. Many databases maintain a transaction log, which could potentially become the performance bottleneck. (4) Figure 9.8 depicts four experiments. The clients in the first two experiments issue `update` queries as required by TPC-B, but those in the last two replace the update

operation by a `select`. (5) Finally, both clients in the first and third experiments issue queries to the same database, as required by TPC-B. In the second and the fourth experiments, the clients issue requests to *two separate database instances*.

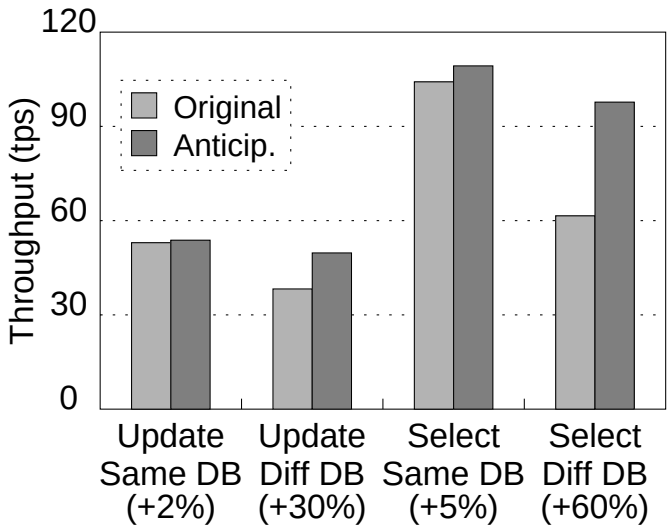


Figure 9.8 : The TPC-B database benchmark and variants: two clients issuing update versus select queries into the same versus different databases.

An update query reads the record first, and then issues an asynchronous delayed write request. The presence of enough delayed writes can give the scheduler more choices, and alleviate the effect of deceptive idleness. Also, seek reduction within a database is severely limited due to almost random queries therein, so the first experiment shows a net improvement of only 2%. The second experiment physically separates the two databases on disk; the impact of anticipatory scheduling is now more pronounced due to seek reduction opportunities within and between databases, and we observe a 30% improvement despite the delayed write requests. Absolute performance is understandably lower than in the first case, due to large seeks between the two

databases. Finally, gains due to anticipatory scheduling are best brought out in the absence of any delayed writes, i.e., when the update operation is reduced to just a select, involving one synchronous read request. We observe throughput improvements by 5% and 60% for requests to the same and different databases respectively.

In summary, our experiments indicate that a database-like workload often stands to gain by the transparent deployment of anticipatory scheduling in the operating system. However, modern commercial databases are highly optimized, and it is likely that they implement some form of application-level prefetching [5]; we have not explored this issue further.

9.7 Proportional-share Scheduling

This experiment demonstrates the impact of the anticipation heuristic for proportional-share schedulers, and the combination heuristic. The workload is chosen to be the fourth TPC-B variant in the database experiment above: `select` operations on different databases, to achieve throughputs of 61 and 98 transactions/sec (i.e., 60% improvement with anticipatory scheduling).

Figure 9.9 depicts an experiment where this workload is subject to proportional scheduling. We use the Stride scheduler augmented with underlying seek reduction, as described in Section 8.6; the relaxation threshold τ is set to 1 second. Proportions of 1:2 are assigned to the two TPC-B clients p and q; these are in terms of disk utilization (not throughput, without loss of generality). In the three cases, the

anticipation framework is either disabled, or separately configured with the Stride or the combination heuristic respectively.

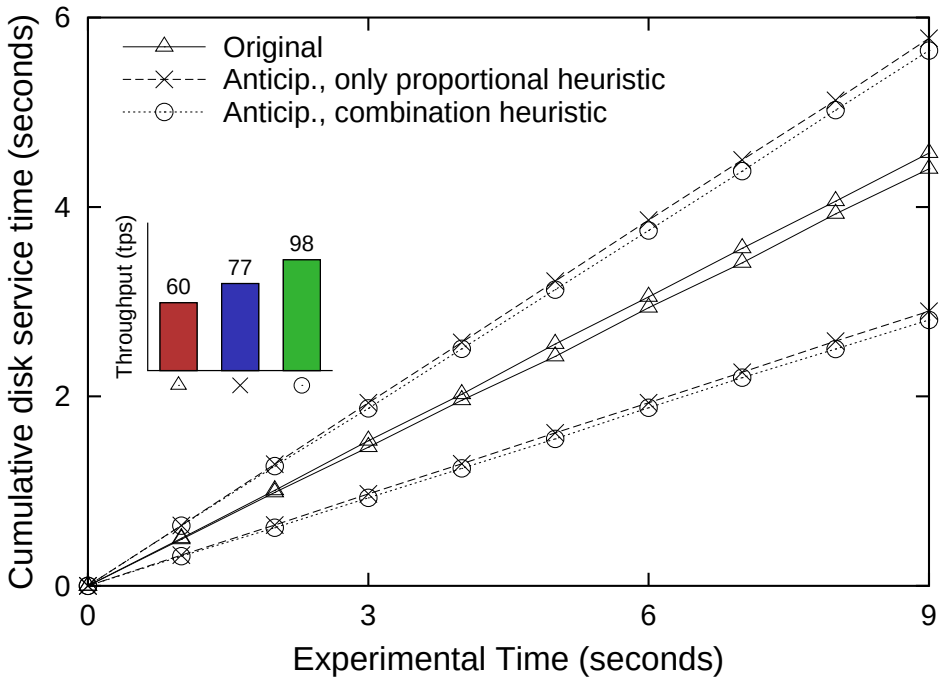


Figure 9.9 : Proportional-share scheduler. Three experiments: (△) original: 1:1 proportions, (×) anticipatory with proportional heuristic: 1:2 proportions, and (○) anticipatory with combination heuristic: 1:2 proportions with maximum throughput.

In the original system, the scheduler always multiplexes between requests from the two processes, and incorrectly achieves proportions of approximately 1:1, with the fairly low throughput of 60 tps. When we turn on anticipatory scheduling with the heuristic for proportional-share schedulers, it realizes that process q (with the higher share) is lagging behind, and waits for it. With average seek and transfer times of 9ms and 3ms, the scheduler manages to achieve 1:2 proportions by servicing 5 requests from q for every request from p. This is sufficient to exploit locality between requests of *one process*, namely q; throughput improves to 77 tps, i.e., by about half

the maximum possible. This results in a corresponding total utilization drop of about 2%, as is seen by utilizations of both processes decreasing proportionally.

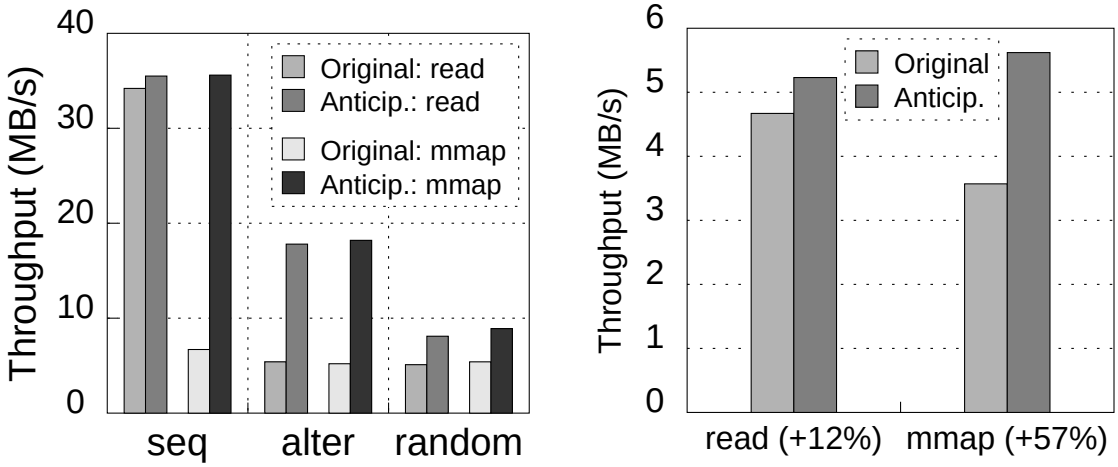
The combination heuristic, on the other hand, realizes the seek reduction potential in waiting for both processes. It thus services several requests from each process, and achieves the full 98 tps throughput, while retaining proportions of 1:2.

9.8 Advanced hardware

We wish to determine the effect of anticipatory scheduling on modern hardware, using the next generation CPUs, disks and controllers. Studies indicate that head seek time improves more slowly than data transfer time; this trend will further aggravate the effects of deceptive idleness. Functionality supported by modern controllers like tagged queueing and improved track buffering and controller-level prefetching may become underused for synchronous I/O. On the other hand, track buffering may assist filesystem prefetching for medium-sized sequentially accessed files, and thus alleviate the problem in some cases. Track buffering also allows the scheduler to wait for the next request, without requiring a complete rotation to read the adjacent sector. On a different note, an increase in CPU speed corresponds to a reduction in application thinktime, which is advantageous for waiting. Thus, a number of tradeoffs can influence the precise gains due to anticipatory scheduling.

To explore this issue, we perform some experiments on an 800MHz Athlon system, with a 15,000 rpm Seagate Cheetah ST318451LW SCSI-3 disk and an Adaptec 19160B

Ultra160 controller. Specifically, we repeat two experiments: the microbenchmark with different access patterns (Section 9.1) and the Apache webserver experiment (Section 9.4). Results are in Figure 9.10.



(a) Microbenchmark (b) Apache webserver

Figure 9.10 : Experiments performed on advanced hardware: 15,000 rpm SCSI disk, 800 MHz CPU.

We note that the maximum bandwidth on this disk is 55% higher than on our original IDE disk, due to a corresponding increase in rotational speed. However, deceptive idleness causes both disks to deliver nearly the same low throughput in the presence of large seeks; this magnifies the best-case gains of anticipatory scheduling to a factor of 5.5, as compared to the earlier factor of 4. Other aspects of this microbenchmark are similar to those on the IDE disk.

Next, consider the Apache webserver experiment. Improvements for the `read` and `mmap` configurations are 12% and 57%. While this is still significant, it is lower than the IDE counterparts. Improved rotational speed, different disk geometry and better

track buffering result in relatively faster servicing of short seeks; these are common in the Apache workload, thus leading to smaller improvements.

To summarize, modern hardware does suffer from deceptive idleness, and stands to gain from anticipatory scheduling. The actual improvements expected on future hardware can be either more or less, depending on precise hardware details and application characteristics.

On a related note, we consider the impact of deceptive idleness and anticipatory scheduling on other disk types, such as redundant arrays of inexpensive disks (RAIDs), just a bunch of disks (JBODs), and network disks. We have not investigated this issue in sufficient depth, but we believe that deceptive idleness can affect such disks, and that anticipatory scheduling can be beneficial. The positioning time estimator would need to derive a useful model of device behaviour, including head positions and redundant copies of data; we believe that this is the key step to adapting anticipatory scheduling to such hardware.

Striped RAID disks, in particular, present an interesting challenge, because reading disks in parallel to read a logical block may involve complications in estimating positioning time. However, we maintain that most of the benefits of anticipatory scheduling can be obtained using a very approximate positioning time model, so this issue should not be significant from a performance standpoint.

9.9 Discussion

This section discusses the practical impact of anticipatory scheduling, and suggests improvements to its design.

9.9.1 Relevance of anticipatory scheduling

Many applications perform non-sequential read I/O on large files, or access many small files colocated on disk, such as those in the same directory. Applications such as webservers and databases often have huge working sets, and issue read requests that cannot be satisfied from memory. This general tendency of applications to issue concurrent, synchronous, non-sequential disk requests has been on the rise [90, 114]. These requests typically do not benefit from traditional filesystem prefetching, and yet possess enough locality to be excellent candidates for seek reduction. This has driven the need for an alternative and more general approach to complement prefetching. Since anticipatory scheduling is based on a much weaker form of prediction, it is feasible in many situations where prefetching is difficult.

Proportional-share schedulers are increasingly gaining prominence in modern systems; for example, they are used in various high-level quality of service systems like using reservation domains to isolate co-hosted websites [22], and performing admission control to guarantee predictable performance of webservers [8]. It is important for these disk schedulers to adhere to their contract; anticipatory scheduling facilitates this for applications issuing synchronous I/O. In practice, proportional-share

disk schedulers will almost always be deployed in combination with a seek reducing scheduler [113]. Our experiments have demonstrated how the combination heuristic brings about simultaneous improvement of both contract adherence and performance.

Real-time disk schedulers (either pure or in combination with seek reducing schedulers) are commonly used to serve and view multimedia content [27, 43]. Under certain circumstances, it is possible for deceptive idleness to cause such schedulers to multiplex between requests from different processes, and consistently violate deadlines. We believe that the anticipatory scheduling framework is applicable to real-time scheduling, but a full exploration of the design and merits of an anticipation heuristic is beyond the scope of this dissertation.

9.9.2 Potential improvements

We suggest two approaches to improve on our proposed design. These are aside from the obvious improvements of making the timing mechanism and the positioning time estimator cheaper and more accurate.

Accumulate more statistics

It is possible for the anticipation heuristic to make suboptimal decisions. We can reduce this chance by augmenting its adaptation mechanism with additional statistics:

(1) Besides tracking expected thinktimes and positioning times, we could collect statistics about the *variance* of these estimates. This gives the heuristic an idea of how accurate these estimates really are. We could then use a technique such as

covariance resetting to discard all previously accumulated statistics whenever this variance becomes too high. (2) The heuristic could keep track of how frequently timeouts expire for each process; if this exceeds some threshold rate, then regardless of all other notions of accuracy, it would know that something is wrong. (3) The positioning time estimator may not be accurate; however, it can measure positioning time after a request has completed service. This provides an indicator for the error in estimation, and thus, our confidence in future decisions. (4) An application might use `aio_read` to issue requests that are actually synchronous; the heuristic can determine this post-facto, and remember it to optimize future decisions.

Relax the two workload assumptions

The anticipatory scheduling framework waits for the process whose request finished last, and collects statistics at a process granularity. Though this is easily the common case, relaxing the assumptions in Section 8.2 can enable the anticipation heuristics to support a wider range of applications, of the following types:

(1) Some proportional-share disk schedulers have a notion of resource principals different from processes, like resource containers [10] and reservation domains [22]. (2) Also, sometimes a group of processes may collectively issue synchronous requests. (3) Applications may simultaneously generate different access patterns on different file descriptors. (4) Some programs may issue two kinds of disk requests from two different parts of the program code, but on the same file descriptor. (5) Seek reduction

intrinsically deals with requests in the same region on the disk; online clustering can classify requests into groups.

To relax the assumptions, the heuristic can collect statistics at all levels of abstraction, i.e., processes, threads, instruction pointer for thread, file descriptors, and disk region – along with their variances. The heuristic can then choose the *highest consistent level* out of these. This has low variance, is expected to be correct, and contains most information.

Chapter 10

Anticipatory Scheduling in Linux

We published a paper on Anticipatory Scheduling in the 18th Symposium of Operating Systems Principles (SOSP 2001) [49], and accompanied it with a FreeBSD implementation. About a year later, Nick Piggin separately implemented it for the Linux 2.5 kernel, where it soon outperformed all other Linux disk schedulers and was made the default scheduler [65, 80, 64, 83, 81, 82].

This chapter tries to summarize three years of experience gained by the Linux community with the anticipatory scheduler. We describe its design, compare it against our FreeBSD prototype to discuss their relative strengths and shortcomings, and present and analyze a number of its benchmarks that were reported on Linux kernel discussion forums.

10.1 Design of the Linux Anticipatory Scheduler

Prior to the anticipatory disk scheduler, Jens Axboe implemented a *deadline disk scheduler* for Linux [9]. Nick Piggin extended this deadline scheduler to incorporate anticipation logic. The anticipatory scheduler in its present form (as of Linux v2.6.13) is composed of an amalgamation of the following four policies, the last of which is a variant of anticipatory scheduling as proposed in this thesis.

10.1.1 Mostly-one-way Elevator

At every step, the Linux disk scheduler performs either a forward seek to choose the next request in front of the head position, or a backward seek to choose the one behind the head position. It mostly prefers the former, and resorts to the latter only if the forward seek is more than twice as long as the backward seek, and if the backward seek is shorter than 1 million sectors. This mostly-one-way elevator scheduling policy avoids starvation, while taking advantage of short backward seeks over long forward ones to improve performance.

10.1.2 Deadlines

If a read request has been pending in the queue for longer than a threshold (125ms), then the scheduler interrupts the current elevator sweep or any anticipation that may be in progress, and seeks to service this lagging request. Similarly, a single write request gets prioritized up if it is held behind a series of reads for more than 250ms. This helps interactive response times to a degree, by ensuring that an individual request does not get held up for too long. However, this time period of 125ms or 250ms is still so large to meet any reasonable performance expectations if there is a dependent chain of requests, which may all be hitting this limit.

10.1.3 Batching

The scheduler batches write requests for up to 128ms before terminating the batch and switching to any held-up read request. Similarly, the scheduler batches read requests for up to 500ms before prioritizing any write request. This also helps interactivity to a degree, by ensuring that an individual read request is not blocked for longer than 128ms behind a flood of write requests; again, 128ms is too long if there is a chain of them.

10.1.4 Read anticipation

If applications issue a stream of dependent synchronous requests, then as described in the previous chapter, the system experiences deceptive idleness. None of the above three scheduling techniques alleviate it.

To solve this problem, the scheduler implements anticipatory scheduling as follows. Similar to what is proposed in this thesis, the scheduler maintains decayed mean profiles of thinktime and seek distance per process. It determines if the process whose read request just completed has (1) a mean thinktime smaller than a fixed threshold of 6.7ms, and (2) a mean seek distance smaller than the seek distance to the current best request in the queue.

If both conditions are met, then the scheduler waits for the *same threshold* of 6.7ms for the process to issue what may be a dependent read request, during which time it does not schedule any disk requests.

10.2 Differences from our design

We identified the following four key differences between this Linux implementation and our FreeBSD prototype.

10.2.1 Underlying scheduling policies

Our proposed anticipation framework can be adapted to any kind of underlying scheduler, and in Chapter 8, we have demonstrated this with Aged-SPTF and YFQ. The Linux implementation deploys anticipation over one specific scheduler, viz. the Deadline scheduler with batching. Although independent of anticipatory scheduling, we briefly contrast Aged-SPTF with the Deadline scheduler to understand any impact these differences may have on relative performance.

While the Mostly-one-way Elevator attempts to balance fairness and performance, SPTF only aims for performance and leaves the fairness to its Aging component. Observations from Section 9.2.2 indicate situations where Aged-SPTF prevents starvation but still introduces too much unfairness for general use. Therefore we, in retrospect, feel inclined to prefer Linux's Mostly-one-way Elevator.

Both schedulers support aging/deadlines. Although they work differently, they have a similar effect on interactive performance. Finally, the Linux disk scheduler's batching seems useful for some kinds of workloads, which the vanilla Aged-SPTF scheduler does not support.

10.2.2 Comparing seek distances with thinktimes

Our proposal for anticipatory scheduling stems off the cost-benefit equation proposed in Section 8.4. The cost of waiting is the process' average thinktime, which is compared against the benefit of waiting, i.e. the difference in positioning times between the current best and the anticipated requests.

$$
\begin{aligned}
&benefit = (calculate\_positioning\_time(Candidate) \\
&\qquad - \text{LP}.expected\_positioning\_time) \\
&\\
&cost = \max(0, \\
&\qquad \text{LP}.expected\_median\_thinktime - elapsed)
\end{aligned}
$$

The Linux implementation, however, lacks a mechanism to translate disk repositioning cost from seek distances to time units. Being therefore unable to compare seek distances against waiting time, the Linux implementation aims to approximate the cost-benefit behavior using the following empirical scheme of anticipating for a request if: (1) the anticipated request is closer *by any amount* to the head position than the current best request, and (2) the average thinktime is smaller than a hardcoded threshold (6.7ms).

Consequently, the Linux implementation fails to do the correct thing in gray areas such as (a) when the anticipated request may be only marginally closer than the current best, but may take almost as much as 6.7ms in arriving, so it may *not* be worth waiting for, or (b) when the anticipated request is some small number of sectors away from the current head position (such as another file in the same directory) but

will arrive very soon after and may therefore be worth waiting for.

The initial cut of our FreeBSD prototype had exactly the same problem, and on exercising with some database and webserver workloads, we saw that it encountered the pitfalls mentioned above. We subsequently retrofitted a simple, measurement-driven disk model to estimate disk positioning time from seek distance (with only 75% accuracy – see Section 8.7). This fixed the performance problems, and also removed most of the awkward "knobs" from our implementation.

The Linux community reports (but does not explain) a performance loss when their anticipatory scheduler implementation is used on database server systems – they recommend turning off anticipation and using the deadline scheduler on such systems. We believe from experience with a similar implementation (but without evidence with the Linux one) that retrofitting seek time estimation to achieve the cost benefit equation may fix this problem.

10.2.3 Waiting for the 95 percentile thinktime

As per our proposal, if the scheduler decides that the benefit of waiting for the anticipated request (from the process whose request finished most recently) exceeds the cost of waiting, then it waits for up to a period of time that corresponds to the 95 percentile thinktime of that process.

$$
\begin{array}{|l|}
\hline
waiting\_duration = \max(0, \\
\qquad \text{LP}.expected\_95percentile\_thinktime - elapsed) \\
\\
return\ (benefit > cost\ ?\ waiting\_duration\ :\ 0) \\
\hline
\end{array}
$$

In contrast, the Linux implementation always waits for a fixed threshold interval (6.7ms) that it checks thinktimes against. This is a double-edged sword, for the following reasons.

It can result in excessive waiting, if processes that are usually quick about issuing the next request occasionally – but not too infrequently – take much longer. This can happen for numerous reasons, including random variations in thinktime, bottlenecks on other resources, differences in user interactions, non-uniform file system layouts, etc., resulting in performance loss.

Our proposal to wait for strictly the 95 percentile thinktime estimate customizes the waiting time to the behavior of the process, with a 95% expectation that it would issue its next request within that time. However, it has the drawback of missing opportunities when the thinktime is slightly longer than the 95 percentile. To compensate for these instances, our implementation waits for up to 1ms more than the 95 percentile thinktime. However, a larger variation would be missed, and Linux's approach of always waiting for up to 6.7ms would have caught it.

The exact nature of this tradeoff depends very much on application behavior, and is therefore difficult to evaluate in the general case.

10.2.4 Read requests versus synchronous requests

While our FreeBSD prototype anticipates all synchronous requests, the Linux implementation restricts itself to the subset that is read requests. This fails to capture dependent chains of synchronous write requests that happen during some metadata operations – though this has a relatively minor effect in Linux because most Linux filesystems implement most metadata operations asynchronously. Two notable exceptions are `unlink` and `truncate`, which are synchronous and sometimes performance critical, so the Linux implementation may need to anticipate them if appropriate.

10.3 Linux Anticipatory Scheduler benchmarks

As the Linux anticipatory scheduler was being developed, Andrew Morton [65] and Nick Piggin [80] made a series of postings to public Linux kernel forums reporting numbers from several micro and macro benchmarks that they ran. In many of these experiments, they found the anticipatory scheduler to outperform all other schedulers by a large margin. In a few experiments, they observed small performance losses over the other schedulers.

This section graphically depicts these numbers for clear visualization, and presents explanations and analyses to supplement the respective forum postings. Note that we have not performed these experiments ourselves — our main contribution is in providing what we hope are some key insights towards better understanding these results.

Refer to Figures 10.1 through 10.9. The four bars in each of these figures represent the following four systems under evaluation:

Elevator	Linux v2.4.21-pre4 with the Elevator disk scheduler.
Deadline	Linux v2.5.61 with Jens Axboe's Deadline disk scheduler.
CFQ	Linux v2.5.61-mm1 with Andrea Arcangeli's Complete Fair Queueing disk scheduler.
AntSched	Linux v2.5.61-mm1 with Nick Piggin's Anticipatory disk scheduler.

Now we describe a series of experiments, and then summarize their findings.

1. Parallel streaming reads: In this experiment, ten 100-megabyte files are read in parallel by separate `cat` processes, and their mean runtime and range of runtimes are plotted in Figure 10.1.

```
for i in $(seq 0 9); do
  time cat 100-meg-file-$i > /dev/null &
done
```

The traditional Elevator scheduler in Linux 2.4 achieves the highest mean throughput, but exhibits huge variance in runtimes between `cat` processes. The Deadline scheduler attains the worst throughput, but all `cat` processes finish at about the same time. The Anticipatory scheduler performs the best of all – it performs almost as well as Elevator in throughput, and has almost as low variance as Deadline.

The file system detects sequential disk accesses for each file, and issues asynchronous prefetch requests to ensure at least one outstanding request in the disk queue per process at all times. Therefore, the best strategy to minimize mean runtime is to completely read each file before switching to the next, which is what Elevator does to

achieve its high throughput, although at the cost of large time-to-service experienced by requests from subsequent files.

The deadline scheduler limits time-to-service for each request, and therefore ends up thrashing in a round-robin fashion between requests from different files, switching after every batch of 16 requests.

The anticipatory scheduler has a looser limit on time-to-service, and therefore does not switch between request streams as often as the deadline scheduler does. It uses its batch_expire thresholds to keep the runtime variance low.

Anticipation itself plays a role at locations where a file is laid out non-contiguously on disk. In such cases, the filesystem becomes unable to issue asynchronous prefetch requests because it is unaware of which file would be read next, or where its block would lie. The anticipatory scheduler does not need such an accurate prediction; it simply waits for the next request to arrive. This experiment uses well laid out, mostly contiguous files; but if they were highly fragmented, then the anticipatory scheduler would stand to gain over Elevator (see Section 9.1 for an example).

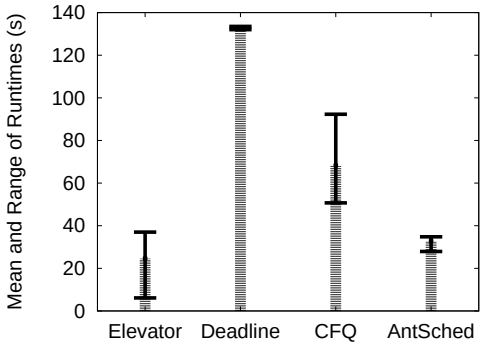


Figure 10.1 : Parallel streaming reads

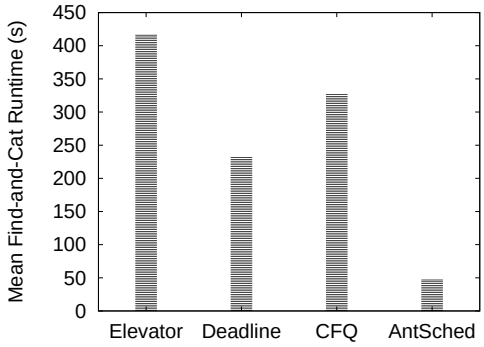


Figure 10.2 : Concurrent reads of many small files

2. Concurrent reads of many small files: This test tries to model the busy web server environment. It has six processes simultaneously reading a number of small files from different parts of the disk. Six separate copies of the linux 2.4.19 kernel tree are set up, and this command is used:

```
for i in 1 2 3 4 5 6; do
  time (find kernel-tree-$i -type f | xargs cat > /dev/null ) &
done
```

As is evident from Figure 10.2, the anticipatory scheduler beats all others hands down on this test. Each stream of disk requests possesses locality as it is issued for a set of files in one directory hierarchy. However, since these files are small, the file system is unable to issue asynchronous prefetch requests for the most part. All other schedulers are therefore forced to thrash between request streams, whereas the anticipatory scheduler exploits the locality by waiting for subsequent requests from the same stream to arrive, and thereby reading many small files from each directory hierarchy at a time.

The anticipatory scheduler services many requests from each stream, while at the same time being fair to request streams by occasionally switching between them, due to the read_expire deadlines built into it. This is clear through the low runtime variance of only 20% between the above jobs.

3. Impact of streaming write on streaming read: This experiment quantifies the impact of a streaming write on streaming read bandwidth. A single streaming write was set up with:

```
while true; do
  dd if=/dev/zero of=foo bs=1M count=512 conv=notrunc
done
```

Figure 10.3 plots the time taken to read a 100 megabyte file from the same filesystem with:

```
time cat 100m-file > /dev/null
```

The anticipatory scheduler outperforms all others by protecting this streaming read from being starved by the flood of writes. It enables the read_expire timeout to function effectively, by waiting for subsequent requests if necessary.

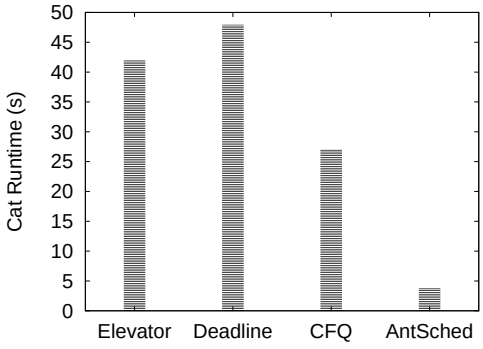


Figure 10.3 : Impact of streaming write on streaming read

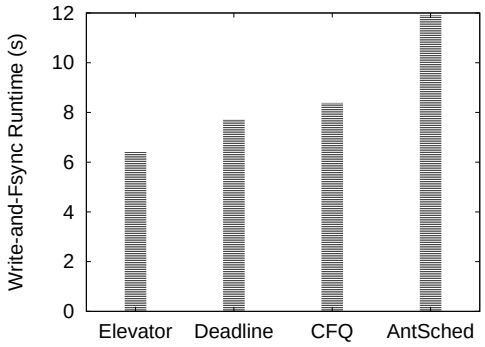


Figure 10.4 : Effect of streaming read on streaming write

4. Effect of streaming read on streaming write: This experiment is the counterpart of the previous one. It quantifies the damage that a streaming read can do to streaming write performance. A streaming read is set up using:

```
while true; do cat 512M-file > /dev/null; done
```

Figure 10.4 measures how long it takes to write out and fsync a 100 megabyte file:

```
time write-and-fsync -f -m 100 outfile
```

While the previous experiment shows that anticipatory scheduling can dramatically benefit the streaming read, this one shows that it can moderately penalize the streaming write as a result. Disk read traffic gets about 2/3rd of the disk bandwidth, making the anticipatory scheduler provide worse service to writes by 100% over the Elevator scheduler, or by 50% over CFQ.

We believe that a real system encountering competition between read and write traffic would generally want to provide better service to the reads, especially if they are synchronous, and especially if the write traffic is for paging I/O or buffer cache flushes. Therefore, we argue that the penalty on write performance is the desired tradeoff for a general-purpose disk scheduler.

5. Impact of streaming write on read-many-files: Figure 10.5 quantifies the impact of a streaming write on operations that read many small files from disk. This experiment, in a sense, fills the gap between experiment 2 and experiment 3. A single streaming write was set up with:

```
while true; do
  dd if=/dev/zero of=foo bs=1M count=512 conv=notrunc
done
```

The following command was used to read all files from a 2.4.19 kernel tree. As a reference, the time to read the kernel tree with no competing I/O is 7.9 seconds.

```
time (find kernel-tree -type f | xargs cat > /dev/null)
```

As Figure 10.5 shows, the anticipatory scheduler outperforms the others by a large margin. By waiting for dependent read requests to arrive, the anticipatory scheduler

does not become forced into servicing the long queue of write requests. Instead, it gets to exploit the seek locality among small files in one directory hierarchy.

All other schedulers suffer from that problem – in particular, the Elevator scheduler gets affected the most because it has no obligation to switch from write to read requests. The deadline and the CFQ schedulers impose expiry times and fairness considerations respectively on the read traffic, and force a single read request to be occasionally serviced. After that happens, they find no more read requests available at that instant, so they switch back to the write requests.

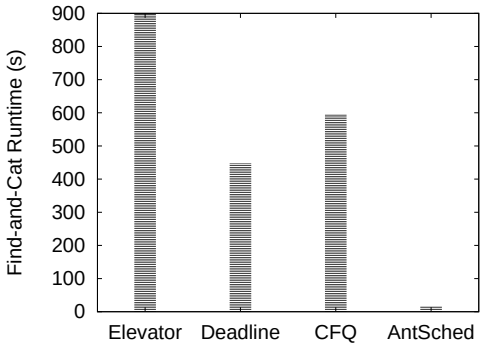


Figure 10.5 : Impact of streaming write on read-many-files

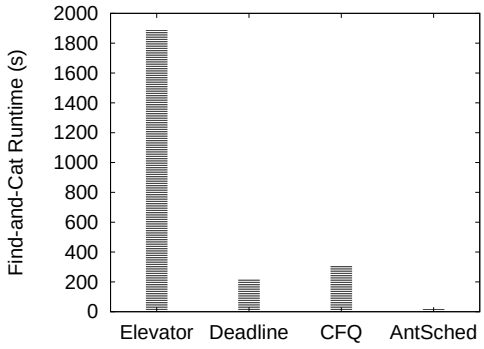


Figure 10.6 : Impact of streaming read on read-many-files

6. Impact of streaming read on read-many-files: This experiment examines the effect that a large streaming read has upon an operation which reads many small files from the same disk. A single streaming read was set up with:

```
while true; do cat 512M-file > /dev/null; done
```

Figure 10.6 depicts the time to read all the files from a 2.4.19 kernel tree using this command:

```
time (find kernel-tree -type f | xargs cat > /dev/null)
```

Once again, the anticipatory scheduler shows a large win, for much the same reasons as the previous experiment. Since the single streaming read induces the file system to trigger asynchronous prefetching, it has the same performance implications as a flood of write requests.

7. Effect of streaming write on interactivity: When a system writes a lot of data to disk, interactive response times often become huge, and user experience drastically degrades. This experiment tries to quantify this effect for different schedulers. It starts a large streaming write, waits for that to reach steady state, then measures how long it will take to pop up an XTerm, and plots this interactive lag time in Figure 10.7.

```
time ssh testbox xterm -e true
```

The three "modern" schedulers clearly perform much better on this test than the original Elevator scheduler, as all three provide some form of support for prioritizing read requests during a flood of write requests. They perform comparably, as this experiment is too rough to bring out subtle differences.

8. Effect of streaming read on interactivity: This experiment sets up a stream of disk read traffic, and times the launch of an XTerm in Figure 10.8.

As in the previous experiment, the three "modern" schedulers perform comparably, and much better than the traditional Elevator scheduler. Minor differences between the three are likely due to random noise.

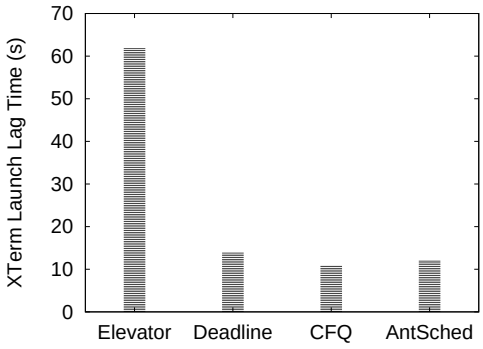


Figure 10.7 : Effect of streaming write on interactivity

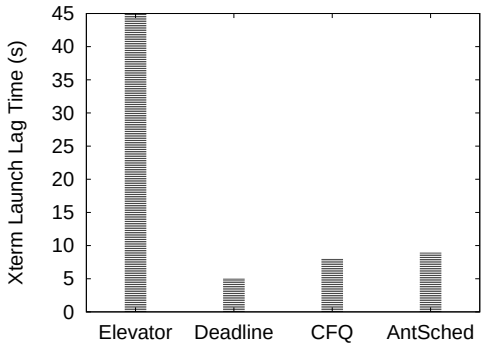


Figure 10.8 : Effect of streaming read on interactivity

9. Time to copy a single large file: Figure 10.9 shows the time taken to copy a large file from one location to another, and then to sync the system.

```
time (cp 1-gig-file foo ; sync)
```

There is no significant difference between the different schedulers.

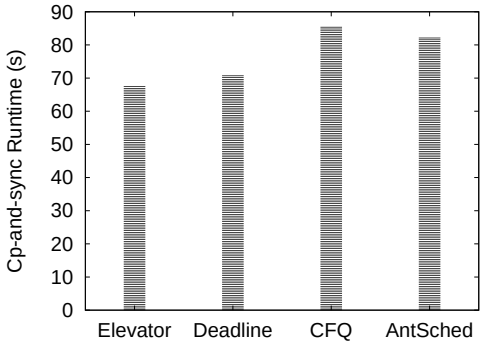


Figure 10.9 : Time to copy a single large file

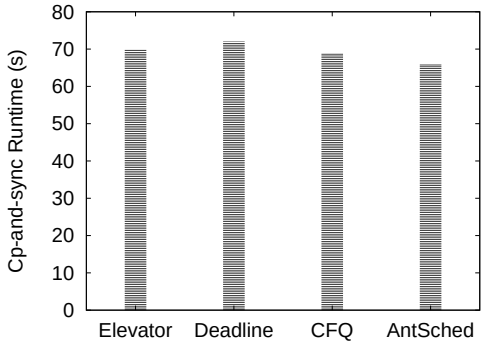


Figure 10.10 : Time to copy many small files

10. Time to copy many small files: This test measures how long it takes to copy a large number of files within the same filesystem. It creates a lot of small, competing read and write I/Os. Three copies of the 2.4.19 kernel tree were placed on

an ext2 filesystem. Figure 10.10 measures the time it takes to copy them all to the same filesystem, and to then sync the system:

```
time (cp -a ./dir-with-three-kernel-trees/ ./new-dir ; sync)
```

The anticipatory scheduler improves runtime by a very small amount, which may just be noise. It could potentially help more by allowing several read operations to happen successively before switching to the write operations, but the current implementation (without VFS support) does not achieve that.

Summary of experiments: The Anticipatory scheduler performs dramatically better than other schedulers in 5 of the 10 experiments, comparably in 4, slightly worse in 1, and dramatically worse in none. It emerges as a clear winner in the choice of a default Linux disk scheduler.

Chapter 11

Related work

The first half of this chapter describes past efforts along three directions that converge into adaptive memory management, namely operating system and application support for memory adaptations, and operating system support for policy control on memory allocation. Then we cover some other virtual memory related APIs in existence, and discuss analogous adaptation schemes in the context of other resources.

The second half of this chapter examines three broad categories of related work in the disk scheduling area. We overview various scheduling algorithms, then move on to techniques that have been proposed to improve disk request handling in the I/O subsystem (external to the scheduler). Finally, we look at scheduling concepts relevant to other domains like CPU scheduling and rate-based flow control.

11.1 Operating system support for memory adaptations

There is a huge body of work in application-controlled page replacement mechanisms and policies [102, 44, 16, 35, 24, 45]. Our work is distinct from the above in that it is orthogonal and complementary to paging. To the best of our knowledge, our work is unique in that it enables applications to dynamically adjust their memory usage based on a metric provided by the kernel that quantifies the degree of memory contention

and enables comparison of the costs of reclaiming memory from different sources. Now we compare our work in more detail against some of these other systems.

In different schemes for *external page cache management* [24, 45], applications are involved in controlling replacement in the operating system file cache. Our work can be viewed as a generalization of this concept to application memory. Our key insight is unique from the above two: Cao, Felten and Li [24] use a two-level scheme for allocating memory in the kernel and deciding replacement policies in applications, whereas Harty and Cheriton [45] take the approach of exposing page faults to applications, whereas we convey a metric that quantifies memory pressure and take independent decisions based on that.

Extensible kernel technology [16, 35] allows applications to modify operating system policies in a flexible manner. As such, sophisticated applications could use their extension interfaces to participate in physical memory management. Our system instead is aimed at enabling application-assisted physical memory management in general-purpose operating systems, via limited kernel changes and few new APIs.

The Sprite operating system maintains a non-unified buffer cache, where the VM and the file cache are disjoint but dynamically resizable [67, 73]. It performs *negotiation* of the VM-FS memory partition, using a technique very similar to our severity metric. Specifically, the VM and FS subsystems maintain the time since last access of the oldest page in each pool, and the page daemon replaces whichever is older. Our contribution can be seen as extending this concept from the file cache to application

memory, providing an application-visible metric for memory pressure, and enhancing it with our *nice* facility.

The Nemesis operating system provides memory isolation via *self-paging* [44], and the VAX operating system performs process-level page replacement [56]. Our system can also provide isolation between processes in terms of memory; in fact, as Chapter 5 explains, we also enable partial isolation through an adjustable self-paging bias. However, our system is primarily aimed at allowing elastic applications to exploit the available physical memory, and the isolation aspect is considered an essential secondary feature.

Recently, the VMware ESX server suggests a technique called *ballooning* to cooperatively manage memory between the operating system and virtual machine guest operating systems for shared hosting [116]. Our system also notifies elastic applications about memory related events, so in that sense there's a similarity; however, we target a mostly cooperative setting in a general-purpose operating system (rather than treat applications as black or gray boxes like in VMware), and our goals and techniques are quite different.

The Mach operating system supports *external paging*, where applications implement custom pagers invoked by the operating system [102]. Our system also performs memory-related notifications, but it is different in that it conveys a severity metric to enables applications to adapt their memory use according to the cost of using this memory.

The Solaris 2.0 operating system provides many features for advanced memory management. For instance, it provides detailed guidelines for manually analyzing memory pressure, by interpreting the output of vmstat, memtool, etc. [62]§7.1.5. It does not automate this memory pressure analysis, nor does it explore application reactions to it. Furthermore, the criteria used do not quantify the age of pages in the system, and are not useful for adaptations.

Our algorithm for managing memory allocations is modelled after the analogy of filling containers with water. There is a strong parallel in the area of network bandwidth allocation, viz. the concept of *max-min fairness* [17, 87, 46]. A bandwidth assignment is max-min fair if all entities receive allocations not exceeding their respective demands, and if increasing the allocation of an entity does not result in the decrease of the allocation of another entity that received an equal or smaller allocation. A popular technique for achieving max-min fair allocation is known as the *water-filling algorithm*, where resources allocations are increased at the same rate to all entities until one saturates, then that one is fixed and the rest are increased further if possible.

This water-filling algorithm is analogous to our approach of allocating memory to applications, viz. by encouraging applications to allocate memory until most of system memory gets used, and then maintaining a state of equilibrium where all applications experience the same level of memory pressure, potentially subject to weighting (see Section 3.3). By drawing the parallel between these two systems, we

are able to inherit some of the machinery for formalizing fairness, and for expressing factors that contribute to it. In particular, our system exports a severity metric and then leaves the details of memory adaptations to the application; instead, we can define some classes of utility functions that certain applications can express, so that the system can take a more active role in understanding and ensuring fairness. Exploring this space further is the subject of future work.

11.2 Application support for memory adaptations

In another direction, the performance-aware application developer community has traditionally experienced frustration from the lack of means for determining how large to set their configuration parameters. They are compelled by portability reasons to treat the operating system essentially as a black box with respect to memory management. This section describes some of these techniques that application developers have adopted to solve or work around this problem.

11.2.1 Database caches

There is considerable literature on the web for manually tuning the size of database caches [14, 111, 85, 61, 108]. Especially while co-locating the database with other applications such as web servers, this size needs to be chosen and periodically re-chosen for best performance.

We now provide an outline of some of the techniques proposed by different database

systems, to emphasize how nearly all such application vendors experience essentially the same problem, which can be solved using our approach.

- The Berkeley DB manual mentions that *choosing a cache size is, unfortunately, an art*. It specifies that the cache must be at least large enough for the database working set plus some overlap for unexpected situations. It describes some heuristics for cache size calculation, such as using the `db_stat` utility to monitor the cache effectiveness.

  It also emphasizes how it is important not to increase the cache size beyond the capabilities of the system, as that will result in reduced performance. Under many operating systems, tying down enough virtual memory will cause memory and potentially the program to be swapped. This is especially likely on systems without unified operating system buffer caches and virtual memory spaces, as the buffer cache was allocated at boot time and so cannot be adjusted based on application requests for large amounts of virtual memory [14].

- A rule of thumb documented for MySQL is to calculate the hit ratio of the cache for each tentative value of the cache size, and keep enlarging it until the hit ratio curve flattens out [111].

- Postgres recommends that the Postgresql shared buffer cache be set to the largest useful size that does not adversely affect other activity [85]. It proposes a tuning methodology by setting 25% of RAM for cache size, and 2-4% for sort

size; these are to be increased if there is no swapping, or decreased to prevent swapping [84].

- The Oracle database (version 9i) makes similar recommendations for manually sizing the cache. In addition, it internally samples hit rate statistics and generates some cache sizing advice. It provides a special view named V$DB_CACHE_ADVICE that predicts the number of physical reads for various buffer cache sizes. After running the database for at least four hours in a production environment, the system administrator is expected to use this prediction to strike a reasonable balance between cache size and hit rate. Essentially, this prediction eliminates the need for the administrator to experiment with several sizes [78].

- Matisse Software Inc. suggests that the Matisse database server be configured with an in-memory page cache sized to at least half of the available system memory for a dedicated back-end server, to speed up the performance of a production system [61].

- Synergistic Office Solutions Inc. sets its database cache to 5MB by default, but they recommend that for best performance, it should be set to 20% or more of the total size of the .DB files in the `\SOS\DATA` folder [108].

With our system, the database can be made to automatically tune its query cache size, by freeing or allocating buffer objects until the oldest cached query result has

slightly higher cost of freeing and regenerating than the cost of paging I/O. In conjunction with this method, the database may also implement some of the conventional wisdom contained in the guidelines mentioned above. Such an autoconfiguration mechanism approximates the balance required for performance, and without the manual intervention.

11.2.2 Virtual machine heap sizes

There are extensive guidelines for tuning JVM garbage collection [107], using very large heaps and intimate shared memory [104], and for setting JVM heap sizes in applications such as the BEA WebLogic server [12]. The rule of thumb is to minimize the time spent performing garbage collection, while maximizing the number of clients it is able to support. Therefore, the heap size may be conservatively set to 80% of system memory. Furthermore, the heap size needs to be manually observed before and after full garbage collections; if the heap size does not change by much, then the heap size setting may be decreased to increase the number of clients or to benefit other applications, without excessively impacting the garbage collection overhead. Garbage collection may also be performed within the young heap (generational garbage collection) for taking advantage of cache locality. Interactive applications may also collect when the user is experiencing periods of idle time. In general, there are two basic strategies for scheduling garbage collections in JVMs: hiding collections when the user is least likely to detect them, or triggering efficient collections when

there is likely to be most garbage to collect [53]§ 7.7.

Our system does not require that these garbage collection heuristics be ignored: on the contrary, generational collection, idle time collection, etc. are usually beneficial. However, whenever these cheap collections prove insufficient, our system triggers full garbage collection upon memory pressure of nontrivial severity. Such full collection can be triggered from Java programs by calling System.gc(). If the JVM is appropriately modified, then any level of garbage collection can be initiated as desired [107]. Similarly, in Microsoft's .NET framework, the programmer can specify the type of garbage collection desired [88, 89]. This choice can be made using heuristics within the virtual machine. Some information from the operating system can assist this decision, such as the distribution of the process' pages within bins of various age ranges. This idea is described in Section 5.3.3.

Thus, with appropriate adaptations triggered by the memory pressure notifications, manual intervention for tuning virtual machine heap sizes can be almost completely made unnecessary.

11.2.3 Soft References

Java provides SoftReference objects [77, 105, 106], which are WeakReference objects that are reclaimed at the garbage collector's discretion. The intention is for garbage collectors to only reclaim them when memory is scarce, and to reclaim them in LRU order of access, so that these objects can be used for caching purposes. Indeed,

there have been mixed reports on the web about experiences with caching using SoftReferences [36].

Many modern JVMs are suboptimal about scheduling the collection of SoftReferenced objects. Some treat SoftReferences as identical to WeakReferences, so that they get reclaimed almost as soon as they lose all the hard references pointing to them, if any. The Sun JVM's garbage collector, by default, keeps a soft reference around one second for every free MB in the heap. If there's 50MB free on the heap, the soft reference will stay around for at least 50 seconds. One can actually configure this ratio using the `-XX:SoftRefLRUPolicyMSPerMB=[ms]` option to the JVM [106].

If support for SoftReferences becomes robust in several JVMs, then it would tentatively seem like an ideal vehicle for implementing application-based caches. The JVM heap size would get automatically configured due to our system of memory pressure notifications, and the SoftReference-based caches can be made to follow suit. However, there is one fatal problem with this. The cost of regenerating the objects pointed to by SoftReferences is fundamentally unknown to the garbage collector. Therefore, when it is taking decisions about whether to shrink its SoftReference pool for a given system-level severity value, it does not know the essential factor of regeneration cost. It would therefore have to estimate the cost, and either conservatively free too much, or optimistically free too little. This and other reasons suggest to us that SoftReferences, in its current form, may never take off as a medium for building memory-intensive and performance-intensive application-level caches. Annotating

SoftReferences with application-defined cost metrics may be necessary to make them more effective.

11.2.4 Web caches

Web caches such as Squid [101] store web objects in files, and retain a memory cache of some size (default 8MB). Note that the filesystem automatically caches some of this file data, so there are three reasons to maintain the additional in-memory cache at the cost of duplicating data in the two caches. These are: (a) the in-memory cache can be controlled using Squid's cache replacement policy; (b) the in-memory cache avoids the read/write system calls for accessing the operating system file cache; and (c) file descriptors become a constrained resource if too many files are simultaneously open, so the in-memory cache avoids having to expend descriptors for the frequently accessed files. Therefore, sizing the Squid in-memory cache using our adaptation system is tricky, because the cost of an in-memory cache miss may be either a file cache access or a disk I/O. This means that our freeing + regeneration cost model for estimating the value of a piece of memory cannot be directly applied to this cache. Instead, the value of these buffers needs to be translated to an approximately equivalent time-cost, after factoring in the above tradeoffs.

Finally, for Squid cache adaptations, there is the issue of internal fragmentation within pages, because application buffers may be smaller than physical pages. If some buffers on a page were recently accessed while most others are old, then the

application may not wish to release such pages when notified of memory pressure. Coupled with the fact that it may have initially allocated a large amount of memory, this application may seem like a memory hog.

This may be unavoidable, since web objects are often small and arbitrarily sized, and padding them up to the nearest page size can lead to considerable wastage of memory. Therefore it becomes essential to pack multiple web objects inside or across physical pages. Therefore, the Squid application may have to be modified to maintain a list of physical pages sorted by time of last access. The replacement policy in times of memory pressure may approximate LRU, by evicting all the objects in the physical page whose oldest object was least recently accessed. Such a deviation in the policy may need to be thoroughly evaluated on realistic workloads. Arguably though, performing such eviction only in times of relatively severe memory pressure, and otherwise using the usual (LRU, GDSF, etc.) policies somewhat conservatively would approximate the conventional Squid behaviour [93].

11.2.5 Other caches

Some other applications attempt to provide internal support for adapting their footprint, and inevitably run up against the lack of system-wide knowledge. For example, as Section 6.5 describes, the Mozilla web browser has a partially developed scaffolding for releasing cache memory from various subsystems (through an internal notification scheme) when there is memory pressure. However, on Unix systems, the only means of

triggering this in Mozilla is to watch for malloc failures – which happen upon exhaustion of virtual memory or upon reaching a resource limit, not on memory pressure. On Windows systems, it also watches for the free memory pool to reach 5%, but this ignores the existence of the file cache, and cannot capture degrees of memory severity. In our system, we can trivially hook such internal memory adaptation schemes to a SIGMEM handler. However, this support in Mozilla is currently immature, so we have not used it.

Performance-critical caches are to be found in many applications. Common applications like Adobe Photoshop caches a large amount of precomputed data. Such applications often manage their own memory to isolate itself from quirks or inabilities of the operating system, mostly as a legacy from the days when it ran on primitive operating systems. However, the drawback of this approach is that Photoshop receives no information from the system as to how much memory it can manage without hurting the rest of the system, thereby aiming for best overall user experience. We believe That our system can help Photoshop arrive at a reasonable answer to this.

OrthoVista is a software product that improves the quality, utility and value of digital image mosaics through radiometric adjustments. The FAQ remarks that *most often, slow performance is the result of an inappropriate value for the memory cache setting* [70]. This leads us to believe that an automatic cache configuration mechanism would be of great value to this product.

Finally, the TNTserver software manual recommends that *if you have questions*

about setting Cache Size, call MicroImages technical support and ask for Ron [110]. Perhaps this helps demonstrate that manually configuring cache sizes can be cumbersome.

11.3 Memory allocation policies in operating systems

Traditionally, physical memory in general-purpose operating systems has been allocated to applications on a per-need basis. Proportional scheduling paradigms for other resources have been adapted to memory [113, 44] in the form of isolation through resource partitioning. SVR4 and BSD4.2 introduced per-process resource limits using the `getrlimit` and `setrlimit` system calls, wherein `RLIMIT_RSS` specifies the maximum resident set size of a process. Linux and FreeBSD implement this by gradually deactivating pages from processes that exceed their resident set limits, so that these pages are preferentially reclaimed later [42, 74]. VMS uses a fixed-size memory partition policy, where each process is assigned a fixed amount of memory called its *resident set*. VMS performs a hybrid of FIFO and LRU replacement at a per-process level, without the use of page reference bits [56, 55, 67].

However, such explicit methods face the obstacle of not knowing the appropriate memory partition size, and having to state it manually based on guesswork or simple measurements, or through some context-specific policy. Such methods for isolation also do not cater to applications' memory demands, since they fail to distinguish between memory allocations for needy versus idle processes. Therefore practical uses of

memory isolation are limited, except as guardians against overusage on shared systems. Their second weakness is the inability to account for memory that is not part of the application's footprint, such as memory in the file cache – accessed through read/write system calls, or mapped and then unmapped (controlling file cache memory is vital for web cache like applications), or kernel memory logically belonging to a process. Our system accounts for all such memory, and even extends nice values to most of this memory.

A resource management framework has recently been proposed for priority-based physical memory allocation [28], and is implemented in the Mungi operating system. This is based on an economic paradigm using concepts like *bank account transactions*, income, taxation, and rent collection. They model applications' need for memory using *resource consumption rates* using the analogy of steadily withdrawing money from a bank account, and prioritize based on that. However, we argue that when cache-like applications compete for memory, but are not allocating any memory in steady state, their rate of consumption is not a good indicator of their need for memory as is the age of the pages in their cache. The system proposed in this thesis has the second advantage that it informs the severity metric to applications, and enables elastic applications to cooperate with the system's memory management efforts.

The VMware ESX server uses memory isolation between virtual machines, but proposes a concept called *idle page tax* [116]. At a high level, it inversely scales the

memory proportions of each virtual machine by a statistically estimated fraction of its pages that have not been accessed for a long time. Thus, the apparent difference of this estimate from our severity metric is that of the fraction of old pages versus the age of the oldest page.

Interestingly, each of the two metrics – idle page tax and age-based severity – is appropriate for its corresponding problem. In the VMware case, the operating system wishes to recover pages from the virtual machine that holds a lot of idle memory. The use of this page may depend on the guest operating system's internal use of the page, so the operating system uses a combination of ballooning to trigger the guest operating system's page replacement policy, and forced pageout in times of severe pressure. On the other hand, our system has the cooperation of applications, and aims to release the *oldest page in the system*, to maintain a balanced level of severity across applications. Beyond that it hands over the choice of freeing the page to the application's policy.

In [8], web servers are adaptively allocated resources to meet a high-level throughput criterion. They are subject to an initial admission control phase, so when they are online, they are partitioned the same amount of memory as initially planned. Over time, their characteristics are studied, and these partitions adjusted through estimations of their working sets.

This non-cooperative scenario does not directly lend itself to our memory adaptation system. Yet, there is use for a partial isolation scheme on the lines of what

Chapter 5 proposes. Then the core working sets of the web servers are retained in memory and isolated from each other, while old memory pages are gracefully shared between all web servers to tolerate bursty workloads and smoothen the periods between partition readjustments. The age-based severity metric is the key to providing partial isolation.

Solaris provides a feature known as *priority paging*, which is mainly applicable to shared code in dynamic libraries [62]§5.8.4. This is completely different from the memory nicing mechanism we propose in this thesis.

11.4 Other virtual memory APIs

There is literature on operating system support for garbage collection, mostly in the form of remapping or unmapping pages that have been swapped out. The Symbolics 3600 is a lisp machine whose virtual memory system performed garbage collection of application data objects. Tru64 and BSD support the `madvise` system call, and AIX provides the `disclaim` system call to have applications inform the operating system of memory that it will not need in the future [53]§ 6.5. Our system provides a different kind of operating system support for garbage collection, geared towards information exchange for heap size management.

Appel and Li [6] identify a set of operating system virtual memory primitives that can be exported to several classes of interested applications for considerable performance benefit. These include concurrent and generational garbage collection, shared

virtual memory, concurrent checkpointing, persistent stores, extending addressability, data-compression paging, and heap overflow detection. These primitives are pertinent to memory protection, whereas our system proposes a primitive that exposes memory pressure.

11.5 Disk Scheduling algorithms

The last thirty years have seen an enormous amount of research in the area of disk scheduling algorithms [96, 119, 20, 48, 52]. The core objective has been to develop scheduling algorithms suited for certain goals, sometimes with provable properties.

11.5.1 Seek-reducing scheduling algorithms

Seek reduction is often crucial for extracting even acceptable throughput for disk intensive applications. The Shortest Positioning-Time First (SPTF) policy greedily aims to improve disk throughput by always scheduling the available request that has the minimum head repositioning overhead time [119]. This policy has two drawbacks: firstly, this policy can incur huge maximum response times due to starvation of old requests. Several variants have been proposed to address this problem. For example, Grouped Shortest Positioning-Time First (GSPTF) divides the disk into cylinder groups, and applies SPTF in each group before moving on to the next [96]. Aged Shortest Positioning-Time First (ASPTF) applies an aging function to the times

computed in SPTF. When a request remains pending for too long, it automatically becomes more likely to get serviced. There are many aging strategies, corresponding to characteristics of the response time bound. ASPTF generally demonstrates remarkable performance, both in terms of high throughput and bounded maximum response time [119, 52, 96].

The second problem with the above algorithms is their requirement for intricate knowledge of the disk layout, in order to predict future head positions and access times accurately. An appreciable component of access time is accountable to rotational latencies, and these are tricky and CPU-intensive to estimate [52, 48]. A popular disk-independent simplification is to schedule requests based on logical block number (LBN) differences, used as an approximation to head repositioning time. Though there can be significant performance loss (up to 18% in practice) in this approach, the remarkable ease and generality of implementation often warrants its deployment.

An example of these LBN-based scheduling policies is Shortest Seek First (SSF), which provides high throughput, but with possible starvation. A popular and widely deployed variant in UNIX-based systems is Circular Look (C-LOOK), also known as the Elevator algorithm. The head traverses the disk in only one direction, servicing all the available requests on its way. On reaching the last such request, the head moves quickly back to the first available request, and starts over. This algorithm is equally fair towards requests directed at various locations on the disk. It avoids starvation, but can incur huge response times (even minutes) if a process several issues

many requests and forces a large portion of the disk to be read [96]. Yet this policy provides fairly good throughput and usually low response time, and is implemented in FreeBSD and Linux.

11.5.2 Proportional-share scheduling algorithms

Quality of service schedulers are increasingly gaining prominence in modern systems. Accurate proportional-share schedulers are required for various high-level quality of service systems, like using reservation domains to isolate co-hosted websites [22], and performing admission control to guarantee predictable performance of webservers [8]. We examine a brief taxonomy of proportional-share scheduling algorithms.

A proportional-share scheduler is expected to allocate resources between resource principals, in proportion to some assigned shares. Generalized processor sharing (GPS) [75] is an idealized "fluid" model that theoretically achieves this goal at all times. However, requests are not infinitesimally divisible, and service can be provided only to one principal at a time. These two intrinsic inexactnesses drive the need for approximations like packet-by-packet GPS (PGPS) [75], Weighted fair queueing (WFQ) [31] and VirtualClock [123] in the network packet scheduling domain. The first two are based on a notion of global virtual time, whereas VirtualClock considers per-stream time (and has the problem of an inactive stream later monopolizing link usage [75]).

Similar proportional-share CPU scheduling policies have been proposed, with sub-

tle differences. Start-time fair queueing (SFQ) [43] is an application of WFQ to a hierarchical multimedia scheduler. The Stride scheduler [117, 115] uses methods similar to VirtualClock, and uses the flexible ticket abstraction to manage allocations. Move-to-rear list scheduling (MTR-LS) [21] simultaneously guarantees cumulative service, delay bound and fairness. (Cumulative service is shown to be more meaningful on a realistic system with multiple resources to manage. In this context, [20] mentions how managing each resource in isolation is not realistic due to "closed-loop" behaviour of most applications). Yet-another fair queueing (YFQ) [20] is a disk scheduling variant to WFQ, and performs novel disk-specific optimizations like batching and overlapping.

Lottery scheduling [118] is a randomized algorithm that allocates resources to principals on a coarse timescale. A unit of resource is assigned to a principal with probability proportional to its assigned share.

11.6 Improved I/O processing outside the scheduling policy

Though scheduling policies have been analyzed in great detail, relatively less attention has gone into examining the *interface* between the scheduler and the rest of the operating system. For example, Seltzer et al. [96] perform a detail simulation study of various seek reducing schedulers, under the premise that a busy system generally has long and bursty disk queues. Our work realizes that these requests may not be of the *right kind*, due to the synchronous manner in which applications generate them.

Deceptive idleness thus originates at this interface, and is closely related to other techniques implemented in the I/O subsystem.

For write requests, IRIX implements *I/O pacing* to prevent programs from saturating the system's I/O facilities. It enforces per-file high and low water marks on the number of queued requests. The low water mark here is to buffer write requests and increase opportunities for seek optimization; this can be viewed as a logical counterpart of Anticipatory Scheduling for write requests.

Also in the context of efficiently handling asynchronous requests, *freeblock scheduling* [60] has been proposed to increase media bandwidth utilization. It dictates head movement paths so as to service asynchronous requests approximately enroute to the synchronous ones. This system bears a logical similarity to Anticipatory Scheduling; both improve the scheduling policy by making use of information about request characteristics, acquired at the scheduler-kernel interface.

Like deceptive idleness, a different kind of scheduler starvation has been noted by [52] in the context of the Aged-SPTF scheduler. The condition arises if aging requests are prioritized abruptly at some age threshold, and if the rate of incoming requests exceeds service rate. Every scheduler choice becomes forced due to prioritization rather than SPTF, and the scheduler degenerates to FCFS. This is a more obvious phenomenon than deceptive idleness, and the solution clearly involves smooth aging.

Our Anticipatory Scheduling solution adaptively determines application behaviour

with respect to synchronous request issue. Understanding application behaviour in various aspects of the disk subsystem can yield significant benefits, and such methods are gaining prominence. For example, adaptive block rearrangement [1] co-locates frequently referenced blocks, thus improving seek performance.

Filesystem prefetching [99] is a well-researched area, and for moderately regular workloads, asynchronous prefetch can hope to transparently eliminate deceptive idleness and bring about improved performance. To improve the feasibility of prefetch, techniques such as transparent hints by speculative execution [76] have been proposed. In comparison, our Anticipatory Scheduling approach generally achieves slightly lower performance than prefetched I/O, but is feasible in more situations due to weaker demands on predictive power.

11.7 Other scheduling domains

We take a brief look at CPU and network interface scheduling, to identify some interesting phenomena related to deceptive idleness.

11.7.1 CPU scheduling

CPU scheduling is preemptible, so there is no analog of deceptive idleness in this domain. There is, however, the equivalent of seek optimization by scheduling requests from the same process. Namely, Ousterhout has proposed *gang scheduling* to reduce context switching overhead, by scheduling groups of threads together [72]. Various

studies of cache affinity on processor scheduling have also been conducted [112].

On a different note, non-work-conserving CPU schedulers of various types have been clearly established as a favourable mechanism for handling bursty workloads. Differentiated, prioritized levels of service in a webserver can be provided by scheduling incoming requests from within the kernel. Even when a CPU is idle, not servicing a low-priority request can prevent it from choking higher-priority requests that arrive soon after. We can enforce stricter priorities under bursty load. [2].

A CPU scheduler can be non-work-conserving in a different way, by keeping a few CPUs idle for handling unexpected and bursty workloads. A detailed simulation-driven analysis [92] has shown its effectiveness for non-scalable workloads, possibly with high variance in arrival rate and execution time.

11.7.2 Network packet scheduling

Rate-based flow control by network packet scheduling has traditionally paced the way for concepts like class-based queueing. This domain is non-preemptible, but there is no real reason to expect deceptive idleness. The high bandwidth-delay product forces applications to always maintain windows of outstanding packets, due to which the scheduler never starves. There is no equivalent of context-switching overhead, so a scheduler has no reason to batch packets.

Interestingly, there is an opposite effect here. Consider WFQ [31] scheduling between say 11 active queues, one of which has a share of 10 and the others have

a share of 1 each. Then WFQ will typically schedule ten requests from the first queue, followed by one request from each of the other queues. This leads to periodic bursts of packets, which is undesirable for network congestion control etc. To remedy this, Worst-case Fair Weighed Fair Queueing (WF^2Q) is a work-conserving scheduler that interleaves packets even more than WFQ does [13]. This can be regarded as an optimization in exactly the opposite direction to Anticipatory Scheduling, which aggregates disk requests for seek optimization.

Non-work-conserving packet schedulers were originally unpopular because of sub-optimal average performance. However, integrated services networks require end-to-end delay bounds, which is difficult to provide under bursty workloads. Zhang and Knightly use non-work-conserving schedulers that hold packets in the network, and simulate the original traffic stream. In effect, this approach substantially protects delay bounds from bursty traffic [121, 122].

Chapter 12

Conclusions

This thesis identifies two problems in general-purpose operating systems that arise due to applications running oblivious of each other and inadvertently contending for system resources such as main memory and disk access. We characterize the effect of this class of problems (which we call *friction* in operating systems) on application performance, fairness, and system robustness. We propose techniques that alleviate these two types of friction, and demonstrate their effectiveness.

We identify the problem of *rigidity in physical memory management* as exhibited by applications that can adapt to memory pressure, but are ignorant of the level of contention for memory. We develop *adaptive memory management*, where we export a memory pressure severity metric that quantifies the cost of using memory, and provides a baseline for applications to adapt against. This results in liberal memory allocations in the common case, to yield significant performance improvements in common memory intensive applications like databases (20% speedup) and Java virtual machines (factor of 2 to 4 speedup). This allows applications release of some of this memory under pressure to ensure paging avoidance and system robustness under load bursts, and reduces the need for footprint configuration in many applications. This also enables superior user control on memory allocations and performance, and

provides operating system level mechanisms to enforce the same.

Secondly, we identify the problem of *deceptive idleness* in the disk subsystem, where applications issuing synchronous disk requests cause the disk head to seek unnecessarily. We propose the *anticipatory disk scheduling framework* as an application-transparent method that injects controlled delays into the disk scheduler. We demonstrate significant performance improvements for common disk-intensive applications such as webservers (30% to 70% speedups), fileservers (8% speedup), databases (2% to 60% speedups), and improves adherence to quality of service objectives for synchronous disk I/O. The anticipatory scheduler has since been deployed in Linux, where it is now the default disk scheduler.

Appendix: Anticipatory Scheduling pseudocode

This appendix summarizes the implementation of the Anticipatory Scheduler, with separate heuristics for the SPTF (and C-LOOK) and Stride schedulers. This does not include pseudocode for measuring disk characteristics and building an approximate positioning time calculator. For even greater detail, please refer to our prototype implementation. The source code is made publicly available on the project webpage [50].

I. Generic Anticipatory Scheduling framework

```
// sched_{enqueue,dequeue,finish,collect_stats,evaluate}
//        are the scheduler-specific functions described later

enum STATE { WAIT = -1, IDLE = 0, BUSY = 1,2,3,... } state = IDLE;
integer pending = 0;
boolean expired = false;

// these four functions should set the higher interrupt priority level
// of the timer (splclock or splhigh), rather than of disk I/O (splbio)

issue(new):
    // gets invoked by upper layers of the kernel
    // state == IDLE or BUSY or WAIT

    if (new.is_read_request) sched_collect_stats(new);
    sched_enqueue(new);
    pending++;

    if (new.is_read_request && (new.is_sync_request || state == WAIT))
        new.proc.blocked = true; // little hope in waiting for this process

    if (state == IDLE || state == WAIT) schedule();


schedule():
    // internal function. sometimes takes a request from the pool and
```

```
    // services it; otherwise just returns with the timeout activated.
    // called with pending > 0

    static request last, next;      // retain values across calls

    if (expired)
    |   // state == WAIT
    |   expired = false;
    |   next = sched_choose(); // need this for SCSI only
    else if (last.proc.blocked)
    |   // state == WAIT
    |   next = sched_choose();
    else
    |   // state == IDLE or WAIT, or BUSY if SCSI
    |   next = sched_choose()
    |   duration = next.is_read_request ?
    |         sched_evaluate(last, next) : 0;
    |   if (duration > 0)
    |   |   if (state == IDLE)
    |   |   |   state = WAIT;
    |   |   |   set_timeout(duration);
    |   |   else
    |   |   |_    // do nothing; in particular, don't retrigger timeout
    |_  |_  return; // "next" contains the chosen candidate

    next = sched_choose();          // probably need this for SCSI
    sched_dequeue(next);
    pending--;
    if (state == WAIT) { cancel_timeout(); state = IDLE; }
    state++;                        // idle->busy, busy->busier
    last = next; next = NULL;
    PERFORM_DISK_IO(last);          // run request on disk


timeout_expire():
    // gets called by timeout code, in interrupt context
    // state == WAIT, pending > 0, next != NULL
    expired = true;
    schedule();


finish(req):
    // gets called by disk driver when current finishes execution
    // state == BUSY
    state--;                        // busy:1->idle, busier->busy
    sched_finish(req);
    req.proc.prev_finish_time = NOW;
    if (pending) schedule();
```

II. Anticipation heuristic for SPTF

```
    sched_collect_stats(new):

        // every process has:
        //    buckets:[0..N] with N = 30 (i.e. least count = 500us)
        //    expected_thinktime, expected_most_thinktime
        //    expected_seek_distance, expected_seek_direction
        //    expected_positioning_time

        if (req \in scheduler_queue
            && req.is_read_request && req.proc == new.proc)
              thinktime = 0
        else
              thinktime = min(15ms, NOW - new.proc.prev_finish_time);
                  // the process issuing "new" computes for some time
                  // between its finish time for previous request and NOW

        loop:i:(0..N) new.proc.bucket[i] *= 0.9         // decay all buckets
        new.proc.bucket[(thinktime/15ms) * N] += 1;     // increment this bucket
        new.proc.expected_thinktime      = median(new.proc.buckets)
        new.proc.expected_most_thinktime = 95percentile(new.proc.buckets)


        seekdist      = ABS(new.location - new.proc.prev_location)
        seekdirection = (new.location > new.proc.prev_location) ? 1 : 0;
        new.proc.prev_location = new.location; // update prev_location

        new.proc.expected_seek_distance *= 0.74
        new.proc.expected_seek_distance += seekdist

        new.proc.expected_seek_direction *= 0.74   // used by C-LOOK heuristic
        new.proc.expected_seek_direction += seekdirection

        new.proc.expected_positioning_time =
              calculate_postime(new.proc.expected_seek_distance)
                // we use an approximation of positioning time based on seek
                // distance; an exact version would need some more arguments


    sched_evaluate(last, next):

        next_seek_distance = ABS(next.location - last.location)
        next_seek_direction = next.location > last.location;
        next_positioning_time = calculate_postime(next_seek_distance)

||<-- ADDITIONAL CHECK FOR THE C-LOOK HEURISTIC ONLY
||
||      if (next_seek_direction == 1 /* ahead */
||      |   && last.proc.expected_seek_direction < 0.3 /* behind */)
```

```
||      |____   return 0                        // proceed immediately
||
||      if (next_seek_direction == 0 /* behind */
||      |   && last.proc.expected_seek_direction > 0.7 /* ahead */)
||      |____   return max(last.proc.expected_most_thinktime - elapsed, 0)

        // time elapsed since we finished servicing previous request.
        // this time needs to be deducted from both expected_thinktimes.
        elapsed = NOW - last.proc.prev_finish_time;
        last_expected_time = last.proc.expected_positioning_time +
                             max(last.proc.expected_thinktime - elapsed, 0)

        if (next_positioning_time < last_expected_time)
        |   return 0; // no point waiting; proceed immediately with "next"
        else
        |   return max(last.proc.expected_most_thinktime - elapsed, 0)
        |     // If we wait for 95 percentile of thinktime, then with 95%
        |     // probability, we would expect to receive the next request
        |____ // before we timeout. This time is bounded by 15ms.
```

III. Anticipation heuristic for Stride

```
    sched_collect_stats(new): reduced version of (or same as) that for SPTF

    sched_evaluate(last, next): // PROCESS VERSION

        minclock = min{ proc[i].clock, where (proc[i].pending > 0) }
        elapsed = NOW - last.proc.prev_finish_time;

        if (last.proc.pending == 0
        |   && last.proc.clock < minclock
        |   && max(last.proc.expected_thinktime - elapsed, 0) < 3ms)
        |      return max(last.proc.expected_thinktime - elapsed, 0)
        else
        |____  return 0;

    sched_evaluate(last, next): // RESOURCE CONTAINER VERSION

        minclock = min{ rc[i].clock, where (rc[i].pending > 0) }
        elapsed = NOW - last.proc.prev_finish_time;

        if (last.proc.rc.pending == 0
        |   && last.proc.rc.clock < minclock
        |   && max(last.proc.expected_thinktime - elapsed, 0) < 3ms)
        |      return max(last.proc.expected_thinktime - elapsed, 0)
        else
        |____  return 0;
```

Bibliography

[1] Sedat Akyurek and Kenneth Salem. Adaptive block rearrangement under UNIX. In *USENIX Summer*, pages 307–321, 1993.

[2] Jussara Almeida, Mihaela Dabu, Anand Manikutty, and Pei Cao. Providing Differentiated Quality-of-Service in Web Hosting Services. In *Proceedings of the Workshop on Internet Server Performance*, Madison, WI, June 1998.

[3] An overview of FreeBSD memory management. http://www.freebsd.org/doc/en_US.ISO8859-1/books/design-44bsd/overview-memory-management.html.

[4] David P. Anderson. BOINC: A system for public-resource computing and storage. In *Proc. of the 5th IEEE/ACM International Workshop on Grid Computing*, November 2004.

[5] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Call graph prefetching for database applications. In *HPCA*, pages 281–, 2001.

[6] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *ASPLOS*, volume 26, pages 96–107, New York, NY, 1991. ACM Press.

[7] M. Aron and P. Druschel. Soft timers: Efficient microsecond software timer support for network processing. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

[8] Mohit Aron, Sitaram Iyer, and Peter Druschel. A resource management framework for predictable Quality of Service in Web servers. Technical Report TR03-

421, Department of Computer Science, Rice University, Houston, TX, July 2003.

[9] Jens Axboe. Linux Kernel: Deadline Scheduler Documentation. `linux_kernel_source/Documentation/block/deadline-iosched.txt`.

[10] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, February 1999.

[11] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 187–196, 1993.

[12] BEA WebLogic Java heap size tuning guide. *http://edocs.bea.com/wls/docs70/perform/ JVMTuning.html*.

[13] J. C. R. Bennett and H Zhang. WF^2Q : Worst-case fair weighted fair queueing. In *Proceedings of IEEE INFOCOM '96*, San Francisco, CA, March 1996.

[14] Berkeley db reference guide: Selecting a cache size. *http://www.sleepycat.com/docs/ref/ am_conf/cachesize.html*.

[15] HTTP log files at the University of California, Berkeley. http://www.cs.berkeley.edu/logs/http/.

[16] Brian N. Bershad et al. SPIN—an extensible microkernel for application-specific operating system services. Technical Report 94-03-03, Dept. of Computer Science and Engineering, University of Washington, February 1994.

[17] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice-Hall, London, 1987.

[18] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. of the IEEE Infocom Conference, Volume 1*, pages 126–134, New York, NY, 1999.

[19] Allan Bricker, Michael Litzkow, and Miron Livny. *Condor Technical Summary, Version 4.1b*, 1992.

[20] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Disk scheduling with Quality of Service guarantees. In *ICMCS, Vol. 2*, pages 400–405, 1999.

[21] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. Move-to-rear list scheduling: a new scheduling algorithm for providing QoS guarantees. In *Proceedings of ACM Multimedia*, Seattle, WA, November 1997.

[22] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. The Eclipse Operating System: Providing quality of service via reservation domains. In *USENIX 1998 Annual Technical Conference*, June 1998.

[23] Jinli Cao. Lecture: Secondary Storage Devices. *http://www.cs.jcu.edu.au/Subjects/cp1500/1999/LectureNotes/lect3-storage/node1.html*.

[24] Pei Cao, Edward W. Felten, and Kai Li. Application-controlled file caching policies. In *Proc. of the USENIX 1994 Summer Conference*, pages 171–182, 1994.

[25] W. B. Carr and J. L. Hennessey. WSClock – A simple and effective algorithm for virtual memory management. In *Proc. of the 8th Symposium on Operating Systems Principles*, pages 87–95, Pacific Grove, CA, December 1981.

[26] M. Castro, P. Druschel, A-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*,

20(8), October 2002.

[27] S. Chen, J. A. Stankovic, J. F. Kurose, and D. Towsley. Performance evaluation of two new disk scheduling algorithms for real-time systems. *Journal of Real-Time Systems*, 3(3):307–336, September 1991.

[28] Kingsley Cheung and Gernot Heiser. A resource management framework for priority-based physical memory allocation. In *Proc. of the 7th Asia-Pacific conference on computer systems architecture*, pages 47–56, Melbourne, Victoria, Australia, 2002.

[29] Transaction Processing Performance Council. TPC Benchmark B Standard Specification Revision 2.0, 1994.

[30] R.C. Daley and J.B. Dennis. Virtual memory, process, and sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, May 1968.

[31] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *Proc. of the SIGCOMM '89 Symposium*, Austin, September 1989.

[32] Peter J. Denning. Virtual memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.

[33] Peter J Denning. Before memory was virtual, June 1996.

[34] Peter J. Denning. Virtual memory. *ACM Computing Surveys (CSUR)*, 28(1):213–216, 1996.

[35] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the 15th ACM Symposium on Operating System Principles*, Copper Mountain, CO, December 1995.

[36] Experience with using SoftReferences for caching. *http://www.cs.rice.edu/~ssiyer/r/mem/refs/crazybob-org.html*.

[37] John Fotheringham. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *Communications of the ACM*, 4(10):435–436, 1961.

[38] FreeBSD malloc library source code. *http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/lib/libc/stdlib/malloc.c*.

[39] FreeBSD malloc manual page. *http://www.gsp.com/cgi-bin/man.cgi?section=3&topic=malloc*.

[40] Peter Galvin and A. Silberschatz. *Operating System Concepts*. Addison-Wesley, Reading, MA, USA, 5th edition, 1998.

[41] Steven Glassman. A caching relay for the world wide web. In *Proc. of the 1st International Conference on the World Wide Web*, CERN, Geneva, Switzerland, May 1994.

[42] Mel Gorman. Understanding the linux memory manager. *http://www.csn.ul.ie/~mel/projects/vm/guide/html/understand/*.

[43] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, October 1996.

[44] Steven M. Hand. Self-paging in the nemesis operating system. In *Proc. of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, February 1999.

[45] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proc. of the 5th Conference on Ar-*

chitectural Support for Programming Languages and Operating Systems, pages 187–197, Boston, MA, October 1992.

[46] Madhusudan Hosaagrahara and Harish Sethu. Max-Min Fairness in Input-Queued Switches. In *ACM SIGCOMM '05 Symposium Poster Session*, Philadelphia, PA, August 2005.

[47] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[48] Lan Huang and Tzi-cker Chiueh. Implementation of a rotation latency sensitive disk scheduler. Technical Report ECSL-TR81, State University of New York, Suny Brook, March 2000.

[49] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada, October 2001.

[50] Sitaram Iyer and Peter Druschel. Anticipatory Scheduling, Deceptive Idleness. Project website:
`http://www.cs.rice.edu/~ssiyer/r/antsched/`, 2001.

[51] Sitaram Iyer and Peter Druschel. The effect of deceptive idleness of disk schedulers. Technical Report CSTR01-379, Rice University, June 2001.

[52] D. Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position, 1991.

[53] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for automatic dynamic memory management*. John Wiley and Sons, Inc., New York, NY,

1996. ISBN 0-471-94148-4.

[54] T. Kelly. Thin-client web access patterns: Measurements from a cache-busting proxy. *Computer Communications*, 25(4):357–366, March 2002.

[55] Kenah and Bate. *Vax/VMS Internals and Data Structures*. Bedford, 1984.

[56] H. M. Levy and P. H. Lipman. Virtual memory management in the VAX/VMS operating system. *Computer*, 15(3):35–41, March 1982.

[57] Sheng Liang. *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Publishing Company, Reading, MA, 1999.

[58] Linux Tutorial: Memory management. http://www.linux-tutorial.info/modules.php?name=Tutorial&pageid=260.

[59] C. Lumb, J. Schindler, and G. Ganger. Freeblock scheduling outside of disk firmware. In *Proc. of the USENIX Conference on File and Storage Technologies*, Monterey, CA, January 2002.

[60] Christopher Lumb, Jiri Schindler, Gregory R. Ganger, Erik Riedel, and David F. Nagle. Towards higher disk head utilization: Extracting "free" bandwidth from busy disk drives. In *Proc. of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 87–102, San Deigo, CA, October 2000.

[61] Matisse Software Inc.: Database Administration. *http://www.fresher.com/developers/faqs/database_admin.html*.

[62] Jim Mauro and Richard McDougall. *Solaris Internals: Core Kernel Architecture*. Pearson Education, 2000. ISBN 0-13-022496-0.

[63] Marshall K. McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, 1996.

[64] Andrew Morton. Linux: Anticipatory I/O Scheuler. `http://www.cs.rice.edu/~ssiyer/r/antsched/antio.html`.

[65] Andrew Morton. Linux: Where the Anticipatory Scheuler Shines. `http://www.cs.rice.edu/~ssiyer/r/antsched/shines.html`.

[66] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan L. Cox. Practical, transparent operating system support for superpages. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[67] Michael Newell Nelson. Virtual memory for the sprite operating system. Technical Report CSD-87-301, University of California, Berkeley, June 1986.

[68] NetBSD forum posting on a malloc performance problem. *http://mail-index.netbsd.org/current-users/2000/05/19/0003.html*.

[69] Elliot Organick. *The MULTICS system; an examination of its structure*. MIT Press, Cambridge, MA, 1972.

[70] Orthovista technical support faq: Performance considerations. *http://www.orthovista.com/ovtechfaq04.html*.

[71] J.K. Ousterhout, H. Da Costa, D. Harrison, J.A. Kunze, M. Kupfer, and J.G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proc. of the 10th ACM Symposium on Operating System Principles*, pages 15–24, Orcas Island, WA, December 1985.

[72] John K. Ousterhout. Scheduling techniques for concurrent systems. In *The 3rd International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[73] John K. Ousterhout, A. R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *Computer*, 21(2):23–36, February 1988.

[74] Page replacement in Linux 2.4 memory management. *http://www.surriel.com/lectures/linux24-vm.html*.

[75] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[76] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proc. of the 15th ACM Symposium on Operating Syst ems Principles*, pages 79–95, Copper Mountain, CO, 1995. ACM Press.

[77] Monica Pawlan. Reference objects and garbage collection. *http://developer.java.sun.com/developer/technicalArticles/ALT/RefObj/*, August 1998.

[78] Performance Impact of the Database Cache: Obtaining Database Cache Size Advice in Oracle9i. *http://purl.oclc.org/NET/ssiyer/oracle2*.

[79] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. of the 1st Workshop on Hot Topics in Networks*, Princeton, NJ, October 2002.

[80] Nick Piggin. Linux: Anticipatory Scheduler Ready for 2.5. `http://www.cs.rice.edu/~ssiyer/r/antsched/ready.html`.

[81] Nick Piggin. Linux Kernel: Anticipatory Scheduler Documentation. `linux_kernel_source/Documentation/block/as-iosched.txt`.

[82] Nick Piggin. Linux Kernel: Anticipatory Scheduler Source Code. `linux_kernel_source/drivers/block/as-iosched.c`.

[83] Nick Piggin. Linux: Tuning the Anticipatory Scheuler. `http://www.cs.rice.edu/~ssiyer/r/antsched/tuning.html`.

[84] POSTGRESQL Shared Buffer Cache: Cache size and Sort size. *http://www.postgresql.org/docs/aw_pgsql_book/hw_performance/node8.html*.

[85] POSTGRESQL Shared Buffer Cache: How Big Is Too Big? *http://www.postgresql.org/docs/aw_pgsql_book/hw_performance/node4.html*.

[86] Pricewatch. http://www.pricewatch.com/, May 2005.

[87] Bozidar Radunovic and Jean-Yves Le Boudec. A Unified Framework for Max-Min and Min-Max Fairness with Applications. Technical report, EPFL, Lausanne, Switzerland, July 2002.

[88] Jeffrey Richter. Automatic memory management in the microsoft .net framework. *MSDN Magazine*, November 2000.

[89] Jeffrey Richter. Automatic memory management in the microsoft .net framework, part 2. *MSDN Magazine*, November 2000.

[90] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *USENIX Annual Technical Conference*, June 2000.

[91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, pages 1–15. Association for Computing Machinery SIGOPS, October 1991.

[92] E. Rosti, E. Smirni, G. Serazzi, and L. W. Dowdy. Analysis of non-work-conserving processor partitioning policies. *Lecture Notes in Computer Science*, 949:165–??, 1995.

[93] Daniella Rosu. Personal communication, May 2003.

[94] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[95] Jose Renato Santos, Koustuv Dasgupta, G. Janakiraman, and Yoshio Turner. Understanding service demand for adaptive allocation of distributed resources. In *Proc. of the IEEE Globecom Global Internet Symposium*, Taipei, Taiwan, November 2002.

[96] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *USENIX Winter Technical Conference*, pages 313–324, Berkeley, CA, 1990.

[97] P. Shenoy and H. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Proc. of ACM SIGMETRICS*, Seattle, WA, June 1998.

[98] Prashant Shenoy and Harrick M. Vin. Cello: A Disk Scheduling Framework for Next-generation Operating Systems. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Madison, WI, June 1998.

[99] E. Shriver, C. Small, and K. Smith. Why does file system prefetching work? In *Proc. of the USENIX 1999 Annual Technical Conference*, pages 71–84, Monterey, CA, June 1999.

[100] Richard L. Sites and Richard T. Weitek. *Alpha Architecture Reference Manual*. Digital Press, Boston, MA, 1998.

[101] Squid. http://squid.nlanr.net/Squid/.

[102] Indira Subramanian. Managing discardable pages with an external pager. In *Proc. of the USENIX Mach Symposium*, November 1991.

[103] D. Sullivan and M. Seltzer. Isolation with flexibility: A resource management framework for central servers. In *USENIX 2000 Annual Technical Conference*, pages 337–350, June 2000.

[104] Sun Microsystems. Big Heaps and Intimate Shared Memory (ISM). *http://java.sun.com/docs/hotspot/ism.html*.

[105] Sun Microsystems. Java(TM) 2 Platform, Standard Edition, v1.2.2 API Specification: Class SoftReference. *http://java.sun.com/products/jdk/1.2/docs/api/java/lang/ref/SoftReference.html*.

[106] Sun Microsystems. The Java HotSpot VM FAQ: What determines when softly referenced objects are flushed? *http://java.sun.com/docs/hotspot/PerformanceFAQ.html#175*.

[107] Sun Microsystems. Tuning Garbage Collection with the 1.3.1 Java Virtual Machine. *http://java.sun.com/docs/hotspot/gc/*.

[108] Synergistic Office Solutions, Inc.: Setting Cache Size on the Database Server. *http://www.sosoft.com/fod/fod_423.htm*.

[109] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[110] TNTserver Administrative Documentation.

http://www.microimages.de/server/serveradmin.htm.

[111] Tuning MySQL Server 4.0 Query cache. *http://www.mysql.com/newsletter/2003-01/a0000000108.html*.

[112] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, pages 26–40. Association for Computing Machinery SIGOPS, October 1991.

[113] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *Proc. of the 8th Conference on*

Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 1998.

[114] Werner Vogels. File system usage in Windows NT 4.0. In *17th ACM SOSP*, June 2000.

[115] C. Waldspurger and W. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical report, MIT/LCS/TM-528, June 1995.

[116] Carl Waldspurger. Memory resource management in VMware ESX server. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[117] Carl A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Managament*. PhD thesis, Massachusetts Institute of Technology, September 1995.

[118] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of the 1st USENIX Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.

[119] B. Worthington, G. Ganger, and Y. Patt. Scheduling algorithms for modern disk drives. In *Sigmetrics*, 1994.

[120] X. Yu, B. Gum, Y. Chen, R. Wang, K. Li, A. Krishnamurthy, and T. Anderson. Trading capacity for performance in a disk array. In *4th Symposium on Operating Systems Design and Implementation*, October 2000.

[121] H. Zhang. Providing end-to-end performance guarantees using non-work-conserving disciplines. *Computer Communications*, 18(10), October 1995.

[122] Hui Zhang and Edward W. Knightly. RCSP and stop-and-go: A comparison of two non-work-conserving disciplines for supporting multimedia communication. *Multimedia Systems*, 4(6):346–356, 1996.

[123] L. Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. *ACM Transactions on Computer Systems*, 9(2):101, May 1991.