BACKGROUND USE OF IDLE RESOURCE CAPACITY

by

Lars René Eggert

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

May 2004

Copyright 2004 Lars René Eggert

Table of Contents

Table of Contents iii

List of Figures ix

Abstract xiv

1. Introduction 1
    - 1.1 Key Issues 3
    - 1.2 Related Approaches 5
    - 1.3 Outline 8
2. Overview 11
    - 2.1 Motivation 15
    - 2.2 Applications and Benefits 18
        - 2.2.1 Prefetching and Caching 18
        - 2.2.2 Network Service 24
        - 2.2.3 Disk Service 27
        - 2.2.4 Application-Layer Uses 29
    - 2.3 Challenges 30
        - 2.3.1 Scheduler Properties 33
        - 2.3.2 System Architecture 36
        - 2.3.3 Preemption Cost 40
        - 2.3.4 Tunability 43
        - 2.3.5 Cache Pollution vs. Preload Effect 44

2.3.6 Isolation of Side Effects 45
2.4 Summary 47
3. Preliminary Work 48
3.1 Application-Level Idletime Networking 49
3.1.1 Application-Level Idletime Mechanisms 51
3.1.2 Server Bottleneck Resource 54
3.1.2.1 10Mb/s Ethernet 55
3.1.2.2 100Mb/s Ethernet 56
3.1.3 Experimental Evaluation 57
3.1.3.1 10Mb/s Ethernet 58
3.1.3.2 100Mb/s Ethernet 61
3.1.4 Discussion 64
3.2 Kernel-Level Idletime Networking 66
3.2.1 Effect of CPU Scheduling on Network Transmissions 67
3.2.1.1 Experimental Setup 69
3.2.1.2 Experimental Evaluation 71
3.2.2 Kernel-Level Idletime Mechanism 73
3.2.2.1 Design Goals 74
3.2.2.2 Kernel-Level Design 76
3.2.2.3 Discussion 79
3.2.3 Experimental Evaluation 80
3.2.3.1 Unlimited Background Load 80

3.2.3.2 Limited Background Load ... 82
3.2.4 Discussion ... 84
3.3 Summary ... 85
4. Idletime Scheduling with Preemption Intervals ... 87
4.1 Formal Specification ... 90
4.1.1 Definitions ... 91
4.1.2 Operations ... 92
4.1.3 Axioms ... 93
4.2 Idletime Properties ... 94
4.2.1 Prioritization ... 95
4.2.1.1 Temporally Shared Resource ... 96
4.2.1.2 Spatially Shared Resource ... 97
4.2.2 Preemptability ... 97
4.2.2.1 Temporally Shared Resources ... 99
4.2.2.2 Spatially Shared Resources ... 99
4.2.3 Relaxed Work Conservation ... 100
4.2.4 Isolation ... 106
4.3 Discussion ... 111
4.3.1 Preemption Interval Length ... 112
4.3.2 Queue Hierarchies ... 117
4.4 Quantitative Analysis ... 120
4.4.1 Performance Expectations ... 126

4.4.2 Disk Drive ... 130

4.4.3 Network Interface ... 132

4.5 Summary ... 135

5. Implementation ... 137

5.1 Scheduler Variants ... 137

5.2 Implementation Overview ... 143

5.3 Idletime Disk Service ... 146

5.4 Idletime Network Service ... 151

5.4.1 Networking Overview ... 151

5.4.2 Prototype Implementation ... 154

5.5 Implementation Considerations ... 157

5.6 Summary ... 159

6. Evaluation ... 160

6.1 Disk Scheduler Evaluation ... 165

6.1.1 Random Access ... 165

6.1.2 Sequential Access ... 170

6.1.3 Discussion ... 174

6.2 Network Scheduler LAN Evaluation ... 176

6.2.1 UDP Foreground Traffic ... 178

6.2.1.1 Foreground UDP vs. Background UDP ... 178

6.2.1.2 Foreground UDP vs. Background TCP ... 180

6.2.2 TCP Foreground Traffic ... 183

6.2.2.1 Foreground TCP vs. Background UDP 183
6.2.2.2 Foreground TCP vs. Background TCP 185
6.3 Network Scheduler WAN Evaluation 187
6.3.1 100Mb/s Baseline 189
6.3.2 100Mb/s with 10ms Delay 192
6.4 Network Scheduler Discussion 196
6.5 Experimental Limitations 198
6.6 Summary 199
7. Discussion 201
7.1 Overview 201
7.1.1 Measured vs. Predicted Performance 203
7.1.1.1 Gigabit LAN 204
7.1.1.2 Disk Drive 205
7.1.2 Effects of Preemption Interval Length 207
7.1.3 Impact of the RTT on Foreground TCP 210
7.1.4 Congestion Control in the Background 211
7.1.5 Effects of Speculative Optimizations 213
7.2 Future Work 214
7.2.1 Idletime Scheduler Extensions 215
7.2.2 Automatic Preemption Interval Tuning 219
7.2.3 Spatially Shared Resources 220
7.2.4 Idletime Networking Improvements 222

7.3 Summary 223
8. Related Work 225
8.1 Realtime Systems 225
8.1.1 Examples 226
8.1.2 Discussion 228
8.2 Idletime Execution 232
8.2.1 Process Migration 233
8.2.2 Data Migration 235
8.2.3 Speculative Execution in Hardware 236
8.2.4 Speculative Execution in Software 238
8.3 Isolation Techniques 239
8.3.1 Database Concurrency Control 241
8.3.2 Discussion 242
8.3.2.1 Processes as Transactions 243
8.3.2.2 Concurrency Control for State Merging 244
8.4 Priority Schemes 247
9. Conclusion 252
Bibliography 256

List of Figures

Figure 1.1. Miniaturization of microprocessors (left, source: Intel) and storage systems (right, source: StorageTek).**Error! Bookmark not defined.**

Figure 2.1. Storage hierarchy. 18

Figure 2.2. Queues involved in network communication over TCP. 37

Figure 2.3. Queues involved in network communication over TCP, showing implicit processing priorities in UNIX. 39

Figure 2.4. Preemption cost due to the presence of idletime use (bottom) compared to the non-idletime case (top). 40

Figure 3.1. HTTP network throughput (left) and server CPU utilization (right) for both 10Mb/s and 100Mb/s Ethernet. 56

Figure 3.2. Normalized median foreground response times (with first and third quartiles) for the baseline case and three different application-level idletime mechanisms over 10Mb/s Ethernet; both under light and heavy foreground load. 60

Figure 3.3. Normalized median foreground response times (with first and third quartiles) for the baseline case and three different application-level idletime mechanisms over 100Mb/s Ethernet; both under light and heavy foreground load. 63

Figure 3.4. Normalized mean throughput with 95% confidence intervals of a foreground sender under unlimited load in the basic case (*No*) and with two CPU-based idletime networking mechanisms (*Nice* and *POSIX*), using TCP (left graph) and UDP (right graph). 70

Figure 3.5. Normalized mean throughput with 95% confidence intervals of a foreground sender under bursty load in the basic case (*No*) and with two CPU-based idletime networking mechanisms (*Nice* and *POSIX*), using TCP (left graph) and UDP (right graph). 72

Figure 3.6. Network stack queuing and processing. 74

Figure 3.7. Normalized mean throughput with 95% confidence intervals of a foreground sender under unlimited load in the basic case (*No*)

and with a kernel-level idletime mechanism (*ITN*), using TCP (left graph) and UDP (right graph). 80

Figure 3.8. Mean foreground (light gray) and background (dark gray) throughputs with 95% confidence intervals under unlimited background load in the baseline case (*Just FG*), without idletime scheduling (*No*), and with a kernel-level idletime networking mechanism (*ITN*). Left graph shows TCP foreground throughputs, right graph UDP foreground throughputs. 81

Figure 3.9. Normalized mean throughput with 95% confidence intervals of a foreground sender under bursty load in the basic case (*No*) and with a kernel-level idletime mechanism (*ITN*), using TCP (left graph) and UDP (right graph). 82

Figure 3.10. Mean foreground (light gray) and background (dark gray) throughputs with 95% confidence intervals under bursty background load in the baseline case (*Just FG*), without idletime scheduling (*No*), and with a kernel-level idletime networking mechanism (*ITN*). Left graph shows TCP foreground throughputs, right graph UDP foreground throughputs. 83

Figure 4.1. Temporally shared resource without (left) and with prioritization (right). 95

Figure 4.2. Spatially shared resource without (left) and with prioritization (right). 96

Figure 4.3. Temporally shared resource without (left) and with preemptability (right). 99

Figure 4.4. Spatially shared resource without (left) and with preemptability (right). 100

Figure 4.5. Scheduler with prioritization and preemptability, incurring preemption overhead. 102

Figure 4.6. Idletime scheduler with prioritization and preemption interval. 104

Figure 4.7. Idletime scheduler with prioritization and short preemption interval, incurring preemption overhead. 105

Figure 4.8. Shared operating system state (left) and virtualized operating system state (right). 106

Figure 4.9. Foreground update and state propagation (left); idletime state commit at finish and propagation (right). 110

Figure 4.10. Bounded preemption cost through preemption intervals. 113

Figure 4.11. Long starvation of idletime processing caused by long preemption intervals. 114

Figure 4.12. Idletime worst-case scenario. 115

Figure 4.13. Operation of priority (left) and idletime schedulers (right) in a hierarchy. 118

Figure 4.14. Analysis configuration. 120

Figure 4.15. Effects of preemption interval length when preemption intervals start at the beginning of foreground requests. 122

Figure 4.16. Surface plot of an example predictor function (left) and its corresponding contour plot (right). 128

Figure 4.17. Overview of the expected performance behavior for an idletime scheduler. 129

Figure 4.18. Baseline read performance of a disk drive without idletime presence. 130

Figure 4.19. Predicted foreground (left) and background (right) performance of an idletime disk scheduler. 131

Figure 4.20. Baseline UDP performance of a network interface without idletime presence. 132

Figure 4.21. Predicted foreground (left) and background (right) performance of an idletime network scheduler with UDP traffic. 133

Figure 4.22. Visualization of the preemption interval range shown in Figure 4.21 (right) compared to the overview from Figure 4.17 (left). 134

Figure 5.1. Idletime state machine; adjacency matrix template. 137

Figure 5.2. Idletime state machine with initial constraints. 138

Figure 5.3. Idletime state machine with additional constraints. 139

Figure 5.4. State machine after third set of constraints. 140

Figure 5.5. Four possible variants of the idletime state machine. 141

Figure 5.6. Variant of the idletime scheduler chosen for implementation. 142

Figure 5.7. FreeBSD kernel I/O overview. Adapted from [MCKUSICK1996]. 144

Figure 5.8. FreeBSD disk I/O processing 147

Figure 5.9. Queuing at different layers in the network stack. 154

Figure 5.10. FreeBSD network stack processing. 156

Figure 5.11. Effects of preemption interval length when preemption intervals start at the beginning of foreground requests 157

Figure 6.1. Experimental setup 160

Figure 6.2. Overview of the expected performance behavior for an idletime scheduler. 163

Figure 6.3. Baseline disk read performance without idletime presence under a random-access workload. 166

Figure 6.4. Measured random-access disk throughput (top row) and latency (bottom row). 168

Figure 6.5. Baseline disk read performance without idletime presence under a sequential-access workload. 170

Figure 6.6. Measured sequential-access disk throughput (top row) and latency (bottom row). 171

Figure 6.7. Baseline UDP performance of a network interface without idletime presence. 176

Figure 6.8. Baseline TCP performance of a network interface without idletime presence. 177

Figure 6.9. Measured 1Gb/s Ethernet UDP/UDP throughput (top row) and latency (bottom row). 179

Figure 6.10. Measured 1Gb/s Ethernet UDP/TCP throughput (top row) and latency (bottom row). ... 182

Figure 6.11. Measured 1Gb/s Ethernet TCP/UDP throughput (top row) and latency (bottom row). ... 184

Figure 6.12. Measured 1Gb/s Ethernet TCP/TCP throughput (top row) and latency (bottom row). ... 186

Figure 6.13. Measured 100Mb/s Ethernet TCP/UDP throughput (top row) and latency (bottom row). ... 190

Figure 6.14. Measured 100Mb/s Ethernet TCP/TCP throughput (top row) and latency (bottom row). ... 191

Figure 6.15. Measured 100Mb/s Ethernet TCP/UDP throughput (top row) and latency (bottom row) with 10ms delay. ... 193

Figure 6.16. Measured 100Mb/s Ethernet TCP/TCP throughput (top row) and latency (bottom row) with 10ms delay. ... 195

Figure 7.1. Relative performance prediction error for the network case. ... 204

Figure 7.2. Relative performance prediction error for the disk case. ... 205

Figure 7.3. Variant of the idletime scheduler chosen for implementation. ... 216

Figure 7.4. Idletime scheduler with relaxed transition into preemption intervals. ... 217

Figure 7.5. Variant of the idletime scheduler chosen for implementation. ... 219

Abstract

This dissertation presents idletime scheduling, an operating system scheduling mechanism for using idle resource capacity in the background without slowing down concurrent foreground use of the system. Executing less important tasks using background capacity increases system efficiency and user-perceived performance. Operating systems fail to support transparent background capacity use: most schedulers provide only fair sharing, which can reduce foreground performance by 50% or more.

The idletime scheduler limits the impact of background use on concurrent foreground processing. It partially relaxes the work conservation principle during short "preemption intervals." It provides a generic mechanism that can accommodate resources with performance characteristics that vary by orders of magnitude. The preemption interval length controls the behavior of the mechanism: short intervals aggressively utilize idle capacity; long intervals reduce the impact on foreground performance.

Experiments with an implementation for idletime network scheduling in FreeBSD maintain over 90% of foreground TCP throughput, while allowing concurrent, high-rate UDP background flows to consume up to 80% link capacity. A disk scheduler implementation maintains 80% of foreground read performance, while enabling concurrent background accesses to reach up to 70% throughput. A quantitative

analysis of idletime scheduling predicts the measured performances with an error below 15%. In both cases, as well as other experiments, the idletime scheduler effectively limits the impact of background use on foreground performance.

1. Introduction

Many computer systems are mostly idle. One study reports an average of 50-70% of the total memory of a cluster of machines to be available [ACHARYA1999], with 15-30 minutes between usage peaks. It concludes, "dips in memory availability (…) are likely to lead to a perception of memory being short." Other studies focus on CPU utilization [MUTKA1987][MUTKA1991][WYCKOFF1998] and report that approximately 70% of the monitored machines in a network were idle.

In the future, idle capacities are likely to keep increasing. Figure 1.1 illustrates the exponential increase in hardware miniaturization for both chip densities (left graph) and storage densities (right graph) over the last decades. It is likely that this trend will continue for at least the near future. In addition, advances in wearable and ubiquitous computing increase the number of computers per person, from one (your desktop) to many: desktop, laptop, PDA, mobile phone, etc. Finally, humans are the bottleneck for many user-interactive workloads. The resource requirements of such workloads are

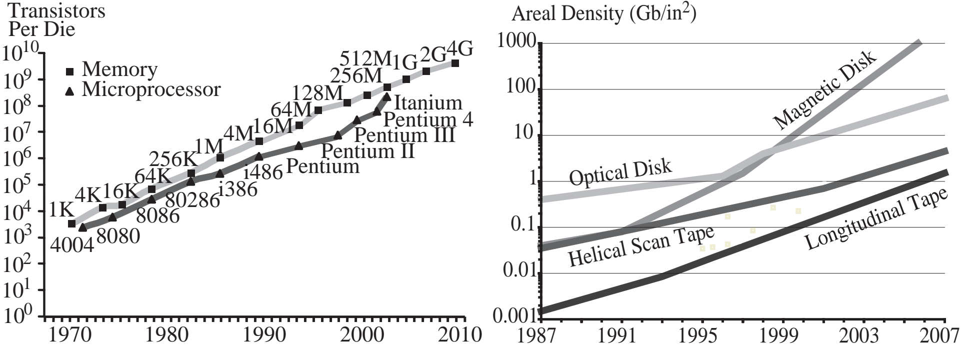


Figure 1.1. Miniaturization of microprocessors (left, source: Intel) and storage systems (right, source: StorageTek)

not likely to increase at the same rates as the underlying hardware. The combination of all these trends will lead to vast quantities of underutilized resources.

This research focuses on the means to utilize such idle capacity for productive background work, without delaying or otherwise interfering with regular foreground processing. For any given workload, a single resource – the bottleneck – limits performance [AMDAHL1967]. Even when the bottleneck is fully loaded, other resources remain partially idle. For example, a system with a fully loaded disk drive may still have significant idle CPU or network capacity. Using this idle capacity productively, without delaying processing at the bottleneck, can improve system efficiency and user-perceived performance.

The key contribution of this work is a general, resource-independent mechanism for background use of idle capacity that minimizes performance impact on foreground use. It establishes background usage as a separate, isolated, low-priority service class, and allows background use of transient idle capacities with less impact on foreground performance. Based on traditional priority queuing, it selectively relaxes the work conservation property for background processing. Unlike many other approaches that require system-wide modifications to support different service levels, the proposed idletime scheduler can establish idletime service as a localized modification to selected schedulers.

Idle resource capacity is therefore an opportunity "to get something for nothing" when utilized for background work. One example is system maintenance tasks such as virus checking or file system optimization. They should execute regularly and can delay user processing due to heavy use of resource capacity. However, it typically matters little exactly when or at which speed such tasks execute. Scheduling them with capacity that is not otherwise in use can eliminate delays for user processing and improve the user-perceived system performance.

Another group of services that will benefit from idletime service is caches and prefetching systems, e.g., prefetching of likely future FTP or web requests [TOUCH1993][TOUCH1994][PADMANABHAN1996]. Conventional prefetchers must explicitly limit their speculative transmissions, to avoid excessive interference with regular network traffic. Idletime use of the network enables aggressive prefetching with minimal, limited interference with regular network traffic. Similarly, idle-capacity use of storage resources (such as memory or disk space) allows the prefetch cache to grow without affecting foreground storage use.

1.1 Key Issues

Ideally, the presence of idletime "background" use in the system should be completely transparent to regular "foreground" processing, in terms of both performance and side effects. For example, the execution time of a given foreground process should be the same with or without concurrent idletime use of some resources. Idletime processing

should fill the “gaps” in resource utilization, without delaying regular use. In such a system, resources could be busy with either foreground or background work at all times, improving overall efficiency.

Furthermore, the side effects of idletime execution must also remain hidden from regular use. For example, when a file system holds idletime data, side effects include the visibility of idletime files to regular processes, and many other pieces of information, such as the count of free disk blocks, or the position of the disk head. To prevent idletime use from interfering with regular processing, the mechanism must carefully hide all such side effects.

In reality, complete isolation of foreground processing from the presence of idletime use is extremely difficult. Interference can occur in terms of both performance and visibility of side effects. Most resources cannot switch between two jobs instantaneously, and switching from idletime use to a new regular foreground job will cause delays. Systems can only avoid these context-switch costs if they enforce reservations by pushing the context switch itself into idle capacity. Resource reservations require extensive application modifications and – when based on worst-case scenarios – can lead to low resource utilization. They may thus fail to support common workloads in a general-purpose system.

Because of the preemption cost associated with stopping the idletime work and starting foreground processing, idletime scheduling can delay regular foreground

processing. Minimizing preemption costs is a key objective for effective idletime scheduling. Overall system efficiency improves only when the gain through idletime use is greater than its associated preemption costs. Even in such cases, preemption costs can still be unacceptable due to user policy. For example, idletime use during a bursty, interactive user task may delay each burst because of the required idletime preemptions. Depending on the user's application, these delays – although small – may be unacceptable.

1.2 Related Approaches

The idea of using idle resource capacity is not new (see Section 8.2). Several systems strive to use idle capacity as part of their regular execution, or use it speculatively in the hopes of reducing future processing times. Process migration systems (e.g., *Condor* [LIZTKOW1988]) and data migration systems (e.g., *SETI@home* [KORPELA2001], *Folding@home* and *Genome@home* [LARSON2002]) attempt to push local computation to idle processors over a network. Other systems exploit idle remote memory as secondary storage [MINNICH1989][NARTEN1992]. Several prefetching and caching systems use idle resource capacity for transfer and storage [AKYÜREK1995][TOUCH1998].

Most existing systems that try to exploit idle capacity do not establish background processing as a separate, lower-priority service class. Instead, they often treat idleness as a system-wide condition and use *ad hoc* schemes to detect it (e.g., CPU utilization

threshold, no user logged in, screen-saver active). During perceived idle periods, they simply add background tasks to the system's workload under the regular execution priority.

For a limited class of applications and workloads, such as the previously mentioned "@home" projects, this coarse approach works surprisingly well. However, it is not a general-purpose mechanism suitable for arbitrary usage scenarios. The approach fails to take advantage of idle capacities that exist even during busy periods and can severely affect foreground performance. Finally, many of these approaches, such as process and data migration systems, focus only on a single resource (usually the CPU), and are ineffective at utilizing idle capacities elsewhere in the system.

The proposed scheduler minimizes delays for regular processing in the presence of idletime use. It is a general, resource-independent mechanism with strict prioritization between regular and idletime use, which can utilize short, transient idle times for background work, even when the bottleneck resource is fully loaded.

The proposed idletime mechanism relaxes the work conservation property for idletime jobs. Work conservation requires that a resource must not remain idle while jobs are waiting for service. The idletime scheduler introduces a time delay, called the *preemption interval*, before switching from regular to idletime processing. When additional foreground jobs appear at the resource during the preemption interval, they immediately receive service. In the absence of a preemption interval (i.e., with simple

priority queues), the higher-priority job would have to wait until the ongoing idletime work finished or was preempted (for resources that support preemption). In either case, delays caused by idletime processing would reduce regular foreground performance. Idletime scheduling amortizes these delays over a burst of regular processing requests, and consequently increases foreground performance compared to a simple priority queue.

The length of the preemption interval controls the impact idletime use has on regular processing. With a short preemption interval, the scheduler is more aggressive in utilizing idle capacity for background use, but permits a higher impact on regular processing. With a longer preemption interval, the impact is lower, but idletime performance also decreases, because a longer preemption interval shortens the usable fraction of an idle period. Changing the length of the preemption interval allows tuning of the mechanism according to user policy and current workload, within limits.

Another feature of the proposed scheduler is that it only requires modifications to key resources, instead of widespread system changes. Conventional priority schemes require modifications to all resource schedulers in a processing hierarchy to support arbitrary workloads. When some schedulers remain unmodified, one of them could control overall system behavior under specific workloads, and effectively disable prioritization. The preemption interval of the proposed scheduler introduces controlled delays for the lower-priority idletime service class. This delay causes the formation of

idletime queues that absorb the scheduling (mis-) decisions of non-idletime schedulers earlier in the processing hierarchy.

1.3 Outline

Chapter 2 will give a detailed overview of the proposed idletime scheduler, and evaluate how it improves existing services and applications, along with enabling new ones. Examples include improving prefetching, precomputation and caching, transparent replication of data, and improving scheduling of maintenance tasks, such as *cron*. Chapter 2 will also discuss challenges in providing idletime service with conventional quality-of-service approaches, and identifies the key issues a successful solution must address, such as minimizing foreground delays due to preemption of idletime work.

Chapter 3 discusses preliminary work in application- and kernel-level mechanisms to establish idletime scheduling of network transmissions. It first evaluates several application-level mechanisms [EGGERT1999] to enable idletime network service for the *Apache* web server [APACHE1995]. The second part of Chapter 3 presents a preliminary kernel-level mechanism for idletime network scheduling, and evaluates its effectiveness experimentally [EGGERT2001A][EGGERT2001B]. Both application- and kernel-level schedulers are effective in establishing different levels of service for network transmissions. However, they fail to completely shield foreground transmissions from the presence of background traffic, and are unable to utilize

significant available network capacity for background use. Furthermore, they only support network scheduling, and do not provide a generic mechanism for arbitrary resources.

Chapter 4 discusses the theoretical principles for idletime scheduling with preemption intervals in detail. It presents a formal model for resource processing, defines its key operations and axioms, and then describes the proposed scheduler in terms of the model. A quantitative analysis of the model results in a simple mathematical model that can predict the global behavior of the new scheduler for specific resources and workloads.

Chapter 5 analyzes several variants of the idletime scheduler that conform to the properties defined in Chapter 4, and identifies one variant for implementation. It discusses the prototype implementation of idletime scheduling for the disk and network subsystems of the FreeBSD operating system, and discusses the prototype's features and limitations.

Chapter 6 discusses experiments that evaluate the performance of the prototype for different network and disk workloads. Chapter 6 discusses and analyses the results of the experimental evaluation, and compares the measured behavior to the predictions based on the processing model of Chapter 4. One principle result is that the simple quantitative model in Chapter 4 can predict the measured performances of the prototype implementation to within 5-15% (borderline cases up to 20%).

Chapter 7 identifies the strengths and weaknesses of the current mechanism and prototype implementation and suggests areas for future improvements and research. Examples include increasing idletime performance by permitting a fixed foreground overhead, and automatically adapting the length of preemption intervals based on the observed scheduler behavior for the current workload.

Chapter 8 covers related work, such as realtime systems, idletime execution, and other speculative techniques. Finally, Chapter 9 summarizes and concludes this work.

2. Overview

This chapter introduces the concept of using available resource capacity in the background, without interfering with regular foreground processing. It starts with a short overview of resource processing that defines terminology used throughout this document. Later sections motivate the idea of idletime use by observing that idle resource capacity has been plentiful, and likely to increase in the future. However, several properties of existing systems make idletime use impractical. Key challenges lie in system architecture, scheduler properties, absence of preemptive resource use, effects on system caches, and interference through processing side effects. Chapter 4 will then introduce a new mechanism for idletime scheduling in current systems that addresses these issues.

A typical computer system contains multiple resources, normally at least a CPU and some main memory. Usually, a system also has some persistent storage devices (e.g., disks), communication devices (e.g., network interfaces, modem), and user I/O devices (e.g., keyboard, display, audio).

The resource use of processes can be modeled as an event stream, where processes generate resource requests to acquire processing capacities (e.g., "read this disk block" for a disk, "send this packet" for the network, or "run me" for the CPU). Resources process these requests in some order, and may generate resource responses (e.g., "here

is the block you wanted" for a disk read request). Note that some requests may result in an implicit response, such as a "run me" request for CPU capacity.

One characteristic common to all resource types is capacity. Processing resource requests uses all or some of the available capacity of a resource. Capacity used to process one request is unavailable to another. If a request requires more capacity than the resource has available at a given time, the resource must delay the request until it has enough capacity available, or it may even reject the request altogether.

One property that categorizes resources is how they allocate their capacity when serving requests: some resources are spatially shared, whereas other are temporally shared.

Spatially shared resources

> Spatially shared devices divide their capacity into allocation units and can serve multiple resource requests concurrently. Processes must typically lease partial capacity before use. Leased capacity becomes available for reuse only after a process explicitly returns it. Storage capacity (e.g., disk space, memory swap space) is an example of a spatially shared resource.

Temporally shared resources

> Temporally shared resources do not subdivide their capacity for concurrent use. Instead, a single process is leased the full resource capacity for a certain (usually fixed) period. The capacity lease is often implicit in submitting the request, and automatically terminates after the request finishes. I/O devices (e.g., network interfaces) and CPUs are examples of temporally shared resources.

It is possible to model spatially shared resources as temporally shared by treating each allocation unit as a separate resource with an unlimited lease time. For example, instead of viewing disk storage capacity as a single spatially shared resource, from another perspective each disk block is a separate temporally shared resource with an infinite lease time. Accordingly, a scheduler for a spatially shared resource could be an extension of one for a temporally shared resource bundle.

Systems may contain multiple, similar resources. For example, on a system with multiple channel-bonded network interfaces, a request can use any available interface. Such a device bundle gains some characteristics of a spatially shared resource, because its components can serve multiple requests simultaneously. Another example is a multiprocessor, where the individual CPUs execute in parallel.

Some physical devices combine aspects of temporally and spatially shared resources. One example is a disk drive. Its storage capacity is spatially shared (different disk

blocks allocated to different processes), whereas its I/O bandwidth is temporally shared: a drive typically serves only a single I/O request at a time. Because these two capacities are separate, complete support for idletime use must consider both aspects. However, idletime support that is limited to one kind of capacity can still be useful. For example, idletime use of disk I/O bandwidth without idletime storage capacity can be useful if a fraction of storage capacity is reserved for idletime use.

User I/O devices (e.g., keyboard, audio) are a special subcategory of temporally shared devices, for which idletime use may not be appropriate. Users explicitly control these devices, and the operating system should treat them as fully utilized and not override the user's scheduling decisions. However, user I/O devices may share an I/O channel (e.g., USB) with other devices. Idletime use of the shared channel capacity is possible if mechanisms treat user I/O requests as foreground use, to prevent delays.

Another special class of resources supports virtualization. Such resources employ various techniques to present a simplified and improved virtual resource to their users. One example is virtual memory, which presents each process with an isolated address space (simplification) that is often larger than the underlying physical memory (improvement) by paging seldom-used parts of address spaces to secondary storage. Another example is RAID disk arrays [PATTERSON1988] and striped network interfaces [TRAW1995]. By transparently bundling the capacity of several physical resources, they increase performance and fault tolerance.

2.1 Motivation

The key idea behind idletime scheduling is to utilize otherwise unused resource capacity productively. Ideally, the presence of idletime processing in the system would be completely undetectable to foreground tasks. Neither performance nor observable side effects of execution would differ, whether idletime tasks were present in the system or not.

Many system resources often have idle capacities. For any given batch workload, a single resource can become the fully utilized system bottleneck that limits performance at any given time [AMDAHL1967]. This means that even during its most busy periods, the other resources of a system have idle capacity. For example, consider the necessary operations to serve a web request. First, the system bottleneck is usually the network interface, receiving the request, followed by the system bus when moving the request from the network interface to main memory. Then the CPU and system bus limit performance when parsing the request. At some point, the disk and system bus become the bottleneck when retrieving the response data, followed again by the CPU and system bus while encoding the response. Finally, the system bus and network interface limit performance when copying the response to the network card and sending the response. The system appears fully loaded, because at each time during the processing of the web request, a bottleneck resource exists that limits performance. However, idle capacities still exist, for example, CPU and network interface could be idle during the disk access.

Even more idle capacities exist on systems with bursty workloads. Such workloads consist of short bursts of activity followed by periods of inactivity. During periods of inactivity, the system is often completely idle. One example of such bursty workloads is interactive applications on user desktops, such as word processing. As an illustration, the world's fastest typists reach sustained speeds of approximately 150 words/minute and 12.5 characters/second. Current CPUs reaching speeds of 3 GHz and higher can execute up to a mean of 240 million instructions between two keystrokes. The potential for idletime use is large.

Several studies have tried to better quantify the amount of idle resource capacity. Some report approximately 70% of monitored workstations in a networked environment had completely idle CPUs, which are temporally shared resources [MUTKA1987][MUTKA1991][WYCKOFF1998]. Others focused on quantifying the available amount of spatially shared capacity, and report 50-70% unused memory in a cluster of machines [ACHARYA1999]. Idle capacities may be even more plentiful than these studies suggest, due to the use of coarse metrics to determine idle times (e.g., "no user logged on," "screen saver active," or "CPU load minimal.") Short, transient idle times may therefore remain undetected due to quantization effects. Furthermore, none of these studies monitored multiple, different resources.

Idletime scheduling proposes to utilize these idle capacities for productive work. The primary characteristic of a successful idletime mechanism is its impact on regular foreground processing. Clearly, an idletime scheme that significantly delays regular

processing is not useful. Users will disable idletime processing if the decrease in system responsiveness for their foreground tasks becomes unacceptable. A second characteristic is the amount of background processing an idletime mechanism can schedule during a given foreground workload – more is better.

Both these metrics are subjective. The design of the idletime mechanism presented in this work assumed that limiting the impact on foreground performance was the critical characteristic, whereas the performance of scheduled idletime work was secondary. This conservative policy can support a wide variety of workloads, especially cases where idletime usage is speculative and predictions about its usefulness are difficult. Idletime schemes for different requirements may require a different approach and are outside the scope of this work.

Idletime use delays foreground processing whenever a resource must preempt ongoing idletime work to schedule an arriving foreground job – or worse, when a resource does not support preemption, and the idletime work must run to completion before the higher-priority foreground job can receive service. Preemption cost is the most important factor contributing to foreground delay.

In a realistic scenario, the preemption operation incurs some overhead, which is the key delay for foreground processing. Section 2.3 discusses preemption cost and other challenges for idletime use. The next section investigates the kinds of performance improvements that idletime use of temporally and spatially shared resources offers.

The final section of this chapter presents the key idea behind the idletime scheduler, which is the main contribution of this work.

2.2 Applications and Benefits

A wide variety of systems will benefit from the availability of idletime service. This section discusses four example areas in idletime use is beneficial: prefetching and caching, disk and network mechanisms, and application-layer uses.

2.2.1 Prefetching and Caching

For spatially shared storage resources, caches are the main idletime application. A system's storage facilities form a hierarchy according to their access delays, as shown in Figure 2.1. Capacity at higher (faster) levels is typically costly and smaller, whereas capacity at lower (slower) levels is large and less expensive. Caching data from lower levels of the hierarchy at higher levels improves performance by reducing access delays. Swapping is the inverse of caching. It pushes data from higher levels into

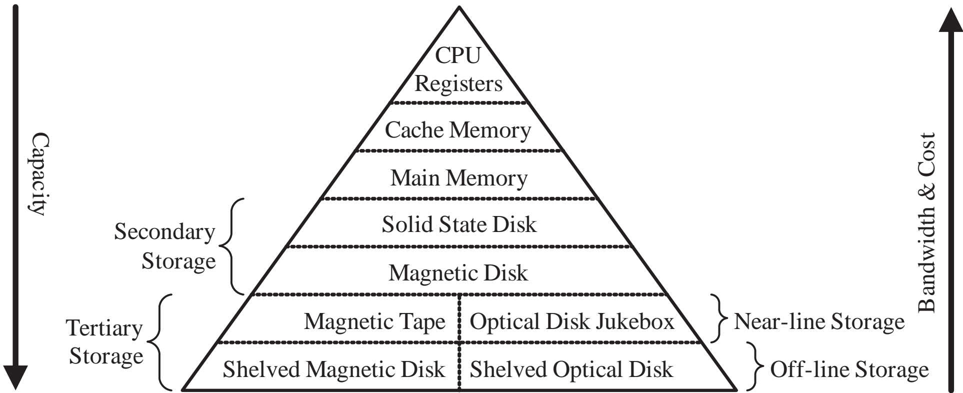


Figure 2.1. Storage hierarchy.

lower ones, to simulate larger virtual capacity at the higher level.

Caches use storage capacity speculatively by replicating data in faster storage to speed up potential future accesses. A cache *hit* occurs when required data is present in the cache, and results in a performance increase due to reduced access time for a data item. The *hit rate* of a cache expresses its effectiveness. Higher hit rates speed up the system by reducing access costs. Increasing the hit rate is a key technique to speeding up overall system processing.

Two separate properties control the hit rate of a cache: the cache size and the cache contents. Obviously, a larger cache will hold more data items, and increases the likelihood of a cache hit. Likewise, optimizing cache content by caching frequently accessed data items will also increase the hit rate. One method of increasing the quality of cache content is by prefetching data items that will likely be accessed in the future. Using idle capacities to support caching can improve system performance.

Maximizing cache size and aggressively prefetching quality data items maximizes hit rates. However, overly large caches can in fact cause a performance decrease by reducing the availability of fast storage capacity for application use. Ideally, a cache should always use any idle capacity available at each level in the storage hierarchy, and grow or shrink in size according to application use. Using idle storage capacity for caching achieves this goal and maximizes hit rates without limiting the availability of fast storage for application use.

Prefetching – to increase the quality of cache contents – also benefits from the availability of idletime use of temporally shared I/O capacity. Without a low-priority, background transfer service, caches must carefully limit their prefetching in order to avoid slowing down concurrent foreground transmission. Issuing prefetch operations in idle I/O capacity enables aggressive prefetching without delaying concurrent foreground use, and thus allows a cache to maximize its content quality, increasing hit rates.

Prefetching and caching are important techniques to speed up execution. The goal is to interleave I/O activity with computation and to prefetch data prior to use, hiding I/O latency. Several proposals focus on prefetching, using different compile-time, run-time and speculative techniques. The remainder of this section will discuss how the availability of idletime service improves existing caching and prefetching systems.

One disk prefetching mechanism uses idle CPU cycles while a process blocks for I/O completion. It speculatively continues execution of a shadow copy of the same process that generates prefetching hints to speed up future I/O [CHANG1999]. The shadow copy executes in a sandbox environment that prevents side effects to become visible to the original process. The authors claim 30-70% reductions in execution times for various benchmarks. This prefetching system already supports partial idletime use by generating disk prefetches with idle CPU cycles. However, the actual disk operations are not issued in the background. They can thus interfere with concurrent foreground use.

Another cache system for out-of-core computations (where large datasets must reside on secondary storage) uses compiler techniques to insert prefetch instructions automatically into the application code [MOWRY1996]. Experiments show that the technique is successful in hiding between 50-98% of the I/O latency, speeding up execution by a factor of 2-3. Similar techniques are also effective for memory cache prefetches [OZAWA1995][MOWRY1998].

Finally, a third mechanism allows applications to disclose future disk accesses to the operating system explicitly by passing hints [PATTERSON1995]. The authors report a performance increase of up to 42% for some applications.

All three approaches strive to identify idle resource times to schedule their prefetches. The first system does so automatically by running the hint generator when the process is blocked. However, due to the absence of prioritized resource access, the hint generator – and the prefetches it generates – can still interfere with regular CPU and disk use by other processes.

The last two systems are even more limited, because they rely on application-level strategies to identify idle times. The mechanisms presented in this proposal could improve and simplify all these systems by shielding regular resource use from the presence of the prefetches, as well as caching speculative data in idle storage capacity.

Successful caching depends on structured information to allow prediction of future usage. Such internal structure may not be readily available, limiting prefetching opportunities. Cross-domain prediction [HUGHES2001][HUGHES2002] utilizes structure in one cache to generate prefetch predictions for others. This increases the usefulness of caches, and can improve performance. One example is speculative execution and caching of likely future DNS queries, based on the contents of a web browser cache. In this example, simulations show a miss rate reduction of 15% with a threefold increase in DNS cache size.

Most operating systems contain caches at many levels in the processing hierarchy (memory cache, disk buffer, ARP, HTTP, etc). Idletime operations can modify cache contents, affecting regular performance. Hint generation for disk block prefetching [CHANG1999], as previously discussed, explicitly tries to pre-load the disk cache with useful data to increase performance.

However, performance decreases due to idletime use can also occur. One study reports a decrease in regular performance when speculatively clearing memory pages. Most operating systems clear memory pages before they allocate them to processes, to prevent exposure of sensitive data, such as passwords. However, page clearing at allocation time severely affects performance. Clearing pages in the kernel's idle loop, such that pre-cleared pages are frequently available at allocation time [DOUGAN1999], may alleviate this problem. However, the authors report that memory cache pollution due to page-clearing limited the overall performance gain. Application-created cache

entries were flushed to store page-clearing data, and application performance therefore decreased, even though page allocations were faster. The obvious fix is to disable cache replacement during idletime processing. Although this decreases the performance of the page-clearer, it retains application state in the cache, and thus improves performance.

Studies investigating other examples of idletime processing, however, find that leaving caches enabled during idletime execution can have a beneficial effect on overall performance, due to a pre-load effect. One such example is the prefetchers discussed previously, which pre-load the disk cache with useful information. Another study investigates the memory cache behavior of a processor with support for multithreading [KWAK1999], and finds that hit rates increase for related threads that exhibit locality-of-reference, whereas they decrease for unrelated threads. A third study monitors the execution behavior of speculatively executing processes [PIERCE1994]. The authors report that although data references increase with speculative execution, data misses increase only moderately; and the prefetch effect more than offsets the performance impact, resulting in improved performance overall.

With extensive idletime use of resources, as proposed in this paper, cache pollution can become an issue. To guarantee isolation, extensions to suspend cache replacements may be required for many of the system caches.

2.2.2 Network Service

One important group of applications focuses on improving user-perceived network service. Idletime use can reduce both connection-open latencies and transmission times. The key idea here is to trade current idle bandwidth for a possible future latency reduction [TOUCH1992]. Ideally, in a network with support for idletime use, lower-priority packet processing will only occur when resources would have been idle in the absence of such traffic.

One well-known application of this idea is web prefetching [TOUCH1993][TOUCH1994][PADMANABHAN1996]. It would greatly benefit from the availability of idle resources use. First, transmitting prefetched data using idle network resources completely shields regular network users from its presence. Consequently, the prefetcher no longer needs to limit its aggressiveness to prevent monopolizing the network bandwidth and potentially delaying concurrent transmissions. Second, larger caches become possible by using idle storage space for the prefetched data.

Even without using idle storage space for caching prefetched information, current idle bandwidth can reduce future network latency by "prefetching the means" [COHEN2000]. This scheme does not prefetch any data, but instead negotiates the means to transfer future data, such as opening TCP connections or resolving DNS names. Idletime execution of these operations creates very little state compared to caching the data, so idletime access to storage capacity may not be necessary.

Another technique described in [COHEN2000] is warming a TCP connection by sending a small amount of throwaway data over a pre-opened connection. This may establish additional state in the end system and router caches, and consequently further improve performance when routers support idletime use and prevent cache trashing due to extensive pre-warming. Using idle network capacity for this purpose improves on the original proposal by permitting a host to pre-send probe packets without interfering with regular traffic. This permits presending of larger amounts of data, allowing TCP to estimate the round-trip time (RTT) and congestion window for the connection better. This may result in better network throughput for later transmissions over the warmed connection, assuming that idletime probes can establish a lower bound for regular network service.

One drawback of existing schemes to warm caches, such as [COHEN2000], is that they suffer from the "tragedy of the commons" effect [HARDIN1968]. The benefit of warming a cache will disappear once the mechanism becomes widely used. Even worse, performance may decrease with aggressive warming due to cache trashing. Idletime execution of speculative operations to warm caches, combined with an idletime-aware cache, may help address this issue.

Execution of two additional network operations during idle time (to "prefetch the means") could be effective. One is speculatively initiated path MTU discoveries [MOGUL1990] to likely future hosts. A PMTU discovery can add one or more round-trip-times to the connection-establishment delay. Hosts supporting PMTU discovery

implicitly perform it whenever they open a TCP connection, and scheduling the discovery during idletime can reduce connection-open delays.

Another similar idea is idletime initiation of IPsec key negotiations using the Internet Key Exchange (IKE) [HARKINS1998] with likely future peers. It also has the potential to save several round-trip times. With current proposals for opportunistic encryption [RICHARDSON2003], IKE negotiations may become much more frequent. Idletime execution of IKE exchanges can reduce user-perceived connection-open delays for successful predictions.

TCP Fast Start speeds up the slow-start period of the TCP protocol [PADMANABHAN1998] by sending more packets than allowed during each congestion window. The authors argue that marking these extra packets with a drop-preference priority eliminates congestion problems. *Rate-Based Pacing* [VISWESWARAIAH1997] is an earlier approach to the related problem of restarting TCP connections after idle periods. Instead of performing a slow-start cycle, *Rate-Based Pacing* resumes transmission at the send rate used prior to the idle period. This may also cause transmission of more packets than allowed by TCP's windowing algorithm. As for *TCP Fast Start*, sending these extra packets in the background can eliminate congestion events.

All previous applications required idle bandwidth to operate. However, even without idle bandwidth, a server system can use idle local resources to increase its network

performance. Most servers (e.g., NFS, FTP, and web) incur packetization cost for each requested object by reading it from the disk (or disk buffer) and splitting it up into a packet chain before transmission. For static objects, caching the packets in idle storage [POSTEL1998] can reduce this cost for repeated accesses [LEVER2000]. As packetization cost increases, e.g., with IPsec authentication, the potential for improvement becomes even greater.

Note that idletime use of the network requires router support. However, the new service model is a simple extension of the current Internet service model, where routers (and hosts) treat packets equally according to a best-effort discipline [CLARK1988]. Idletime use does not change this fundamental model: the network may still reorder, drop, or duplicate packets. Idletime networking is strictly a per-hop function of giving higher processing preference to regular, foreground packets. Section 5.4 discusses idletime networking in more detail.

2.2.3 Disk Service

All the applications for idletime use of available resource capacity described in the previous sections mainly use idle network bandwidth, and to a lesser degree memory and CPU. Idletime use of disk capacity (both I/O bandwidth and disk space) has also the potential to improve system performance.

Most file systems already use read-ahead techniques to improve input performance [PATTERSON1995]. A straightforward improvement would be to execute read-ahead

prefetches (which are speculative by nature) with idle disk resources, and move the disk buffer that caches them into idle physical memory. Prefetches would then no longer interfere with regular read operations, and large idle-memory disk buffer sizes would not limit memory availability for regular uses.

Another technique that would benefit from the availability of idletime disk service is disk block replication [AKYÜREK1995]. This approach spreads replicas of frequently used disk blocks out over the entire disk. In effect, it moves the data closer to the disk arm, reducing arm movement and therefore access times. One drawback of this scheme is that replicas decrease available disk space, and replica management uses disk bandwidth. Using idle disk space and bandwidth would mitigate these shortcomings.

The inverse of the previous scheme is to move the disk arm near spots of likely future accesses during idle time [KING1990][MUMOLO1999]. Unlike the disk block replicators, this approach does not transfer or cache any data, so the memory and disk subsystems need not support idletime use. The drawback is that replication can have better prediction rates than head relocation, because the likelihood of the arm being near the data increases with the replication factor.

Prefetching and caching file system meta-data is another technique to increase file system performance [MOLANO1998]. As with many caches, choosing the correct size is critical for system performance. Using idle memory for the cache solves this

problem, as the cache will automatically shrink as memory use by regular processes increases.

Many file systems must be periodically checked for inconsistencies due to loss of power, etc. One improvement to the Berkeley Fast File System [MCKUSICK1984] is a background daemon that continually monitors and fixes file systems for inconsistencies [MCKUSICK2002]. Such a process would be a prime candidate for execution with idle CPU and disk resources. Similarly, adaptive techniques to optimize performance of log-structured file systems require periodic reorganization of disk contents [MATTHEWS1997]. Executing these tasks with idle resources could improve overall system performance by minimizing interference with regular use.

2.2.4 Application-Layer Uses

Application-layer uses for idle resources also exist. One such application is an improved *nice* utility to schedule periodic optional maintenance tasks in a system. Examples of such tasks are checking for viruses, defragmenting the file system, and auditing system security.

Non-optional system management tasks, typically run from calendar-based schedulers such as *cron* [REZNICK1993], also benefit from using idletime resources. *Cron* runs specified tasks at certain times. Simply running *cron* using idle resources is not sufficient, because regular resource use could then prevent scheduled *cron* tasks to miss deadlines.

However, many tasks scheduled via *cron* do not have absolute deadlines. Guaranteeing that such tasks execute within a certain time interval – as opposed to at a specific point in time – extends the *cron* scheduling model to support use of idle capacity. For example, instead of scheduling a regular disk cleanup explicitly at 2am (because resources tend to be idle at that time of the day), the system would schedule an idletime disk cleanup anytime between 1-2am. If the task did not run by 2am due to unavailable idle resources, it would then execute using regular resource capacity.

Under this model, foreground processing can be isolated from the presence of periodic background tasks by pushing those into idle periods before a deadline. If insufficient idle capacity is available before the deadline, the system switches a *cron* task over to foreground execution. In consequence, operation in the fallback case is similar to regular *cron*, while still isolating regular use when sufficient idle capacities are available.

2.3 Challenges

The previous section gave an overview of the kinds of current services and applications that would benefit from idletime use. However, many existing systems cannot support such idletime use without delaying regular foreground processing. This section will discuss the key challenges for idletime use. Chapter 4 then introduces an idletime scheduling mechanism that addresses these challenges.

Several issues affect the feasibility and effectiveness of a mechanism to use idle resource capacity. The most obvious is the distribution of idle times for a system's resources for a given workload. When idle capacities are rare or very short – i.e., with mostly utilized resources – the chance for performance improvement is low. As previously mentioned, however, ample idle capacity is often available.

Because idletime scheduling introduces a new lower-priority service class, request prioritization is clearly a required property of any idletime scheduler. Resources must give preferential treatment to queued foreground requests and serve them before any waiting idletime requests. Otherwise, foreground performance will decrease. For example, if a network interface starts sending an idletime packet even though a regular packet is also enqueued, it reduces foreground performance. Section 2.3.1 will discuss this issue in detail.

In addition to simple queue schedulers, many systems contain implicit scheduling decisions that can interfere with idletime use. One example is the implicit prioritization between low-level device interrupt processing, higher kernel levels and user-space processing. Interrupt handling on behalf of idletime use can delay higher-level foreground processing, even with priority queues. Section 2.3.2 looks at these cases.

In addition to service prioritization, resources with support for idletime scheduling should also support request preemption. In the ideal case, a resource can immediately

preempt ongoing idletime processing to free capacity for a newly arriving foreground request. For example, consider a 10GB disk device that contains 5GB foreground data and has allocated another 4GB for idletime processing. When more than 1GB of new foreground data needs to be stored, the disk needs to free enough idletime space transparently to accommodate the new foreground data.

In a realistic scenario, these preemption operations incur an overhead, and are the key factor affecting delay for foreground processing. In the previous example, the foreground request requiring additional storage capacity must wait until the resource has reclaimed enough idletime space. If no idletime use were present in the system, this delay would not occur. Likewise, if the reclamation happened instantaneously, the foreground request would also not have to wait.

Although some resources support request preemption, others do not. For example, on the PC architecture, resource I/O operations involving direct memory access (DMA) are not preemptable. When idle capacity on such resources is used for background work, the resource may have to delay newly arriving foreground requests until finished idletime requests free enough capacity to service them. Because service times are typically much longer than preemption times, idletime use of resources without preemption can severely affect foreground performance without additional mechanisms. Section 2.3.3 investigates preemption costs in detail.

The key contribution of this research is a scheduling mechanism that can limit preemption costs in such cases. As discussed in detail in Chapter 4, it relaxes the work conservation property and introduces a preemption interval during which no idletime processing may begin, even if the resource has idle capacity. The length of a preemption interval can vary, and allows adaptation of the overhead of idletime use to a given delay policy.

Finally, idletime use may not be appropriate in all scenarios. Although it has the potential to increase system performance, utilizing resources during otherwise idle periods increases the power consumption of the system, and causes additional wear-and-tear for hardware components. The current idletime mechanism does not address such scenarios. Future idletime extensions may incorporate design ideas from power-aware schedulers [PAPATHANASIOU2003][ZHENG2003] to enable idletime use in restricted environments.

2.3.1 Scheduler Properties

To differentiate between regular foreground and idletime background processing, the operating system needs to associate a flag with each resource request and its corresponding response to indicate service class. Schedulers must honor the priority flag and order their work queues accordingly. Furthermore, resources must implement a preemption mechanism for idletime use when possible, to minimize impact on foreground use.

Many traditional operating systems do not support transparent background use of idle resources. Their resource schedulers usually seek to establish some degree of fairness between their different processes – or at least prevent starvation. This property effectively prevents idletime use, because background requests should not receive a fair share of resource capacity at the expense of foreground processing – they should not receive any share in that case. Background use of idle capacity on such resources can delay or even prevent regular foreground use, because they start background work while foreground work is waiting, or fail to preempt background work when foreground work arrives.

One example of a problematic scheduler is the multilevel feedback queue that many UNIX systems use for CPU scheduling. This variant of a round robin scheduler favors bursty processes, which do not fully utilize their allocated CPU quantum, by raising their priority over time. Additionally, it punishes compute-bound processes by lowering their priorities. Most I/O-bound processes are bursty – they block during device operations – and consequently achieve high CPU priorities. Commonly, the CPU scheduler offers the user processes some degree of control over their priorities. Non-privileged processes may lower their priority from the default, whereas increasing the priority is restricted to privileged processes. However, monopolizing the CPU through this mechanism is impossible; it merely adjusts the share of processing time and does not establish total priority.

Simple first-in-first-out (FIFO) schedulers control access to many other system resources. Although FIFOs by themselves do not assure fairness, they do so in combination with a fairness-enforcing CPU scheduler such as the one previously described, because a process cannot issue any resource requests without a CPU to run on.

Neither FIFO nor round robin schedulers satisfy the prioritization principle; all requests receive equal service. Even a multilevel feedback queue only allows adjustment of shares, and does not allow starvation. To support non-interfering idletime use, resource schedulers must instead augment such scheduling disciplines with priority queues with two service classes for regular and idletime requests.

The CPU scheduler on many POSIX-compliant systems [POSIX1993] already offers this capability. The POSIX CPU scheduler has three distinct priority classes (realtime, regular and idletime), each managed by its own multilevel-feedback queue, and supports preemption. Consequently, processes running under the POSIX idletime scheduling class will not receive any CPU time while processes in higher classes are runnable. Starvation of lower-class processes occurs when higher-class load increases to saturation. Experiments with the POSIX scheduler show that it can isolate regular use from idletime requests to within 1% of overall throughput. Some other experimental CPU schedulers also support an idletime processing class explicitly [FORD1996].

2.3.2 System Architecture

Computer systems contain many queues with their associated schedulers. Some manage physical devices, such as CPUs, memory or network interfaces. Many others are internal to the operating system, and manage buffer pools, multiplexers and work queues. A third kind of scheduler is implicit in the distributed cooperation of different pieces of code that interact according to a certain protocol, such as giving processing priority to interrupt handling.

Processing inside an operating system can be modeled as a directed graph, in which nodes correspond to resources and arcs denote producer/consumer relations. Processes are producer nodes that create resource requests and physical resources are consumer nodes that sink them. A request processed at a resource node may generate zero or more associated requests that in turn propagate to other resources for processing. To execute a specific processing step, a process will issue requests that flow as a request stream through a succession of queues managed by various schedulers before terminating at a physical resource. Loops are specifically allowed and are resolved by the processing rules.

As an example of such a queue hierarchy, Figure 2.2 shows the queues involved in network communication using TCP. When sending TCP traffic (top queue chain in Figure 2.2), data flows from the send buffer of an application (on the left) into the socket send buffer in the kernel. TCP's congestion control algorithm then places the corresponding IP packets into the device send queue, and the network driver (on the

far right) finally transfers these packet to the network interface. TCP inbound processing (bottom chain in Figure 2.2) is similar.

Some operating systems, such as *Scout* [MOSBERGER1996], strictly implement this path-based graph model. A central scheduler manages all paths in the queue hierarchy, and decides which path to service at any given time. Adding idletime support to such systems is straightforward through an extension of the central scheduler. However, the major drawback of *Scout* and similar systems is their vastly different API, which requires application modifications and thus severely limits the usefulness of such an extension.

Idletime support for more conventional operating systems, such as UNIX-based ones, is much more complicated. They too consist of a large number of resources and schedulers that require modification, but additionally have implicit processing rules that interfere with idletime use. One example of such implicit prioritization is giving higher priority to internal kernel processing and preempting process execution for events such as interrupt handling. Even inside the kernel, lower-level events such as

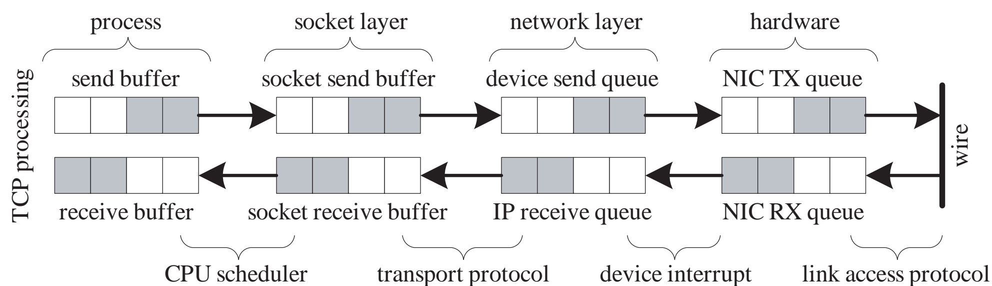


Figure 2.2. Queues involved in network communication over TCP.

hardware interrupts take priority over higher-level system call processing. Furthermore, kernel processing is often work conserving. The operating system will service all pending events before resuming execution of user processes. This can cause priority inversion, where a higher-priority request must wait for the completion of a lower-priority one [LAMPSON1980].

These processing properties interfere with idletime resource use. Giving higher preference to kernel processing can counteract prioritization, because the system may interrupt execution of a foreground process in order to service a kernel event associated with background processing. Furthermore, work conservation will drain existing queue contents before giving processes a chance to schedule more work. A series of queued background requests at a lower level may receive service while a foreground process must wait to schedule more requests at a higher level. This effectively disables prioritization, even with priority queues.

Figure 2.3 shows the implicit priorities in the queue hierarchy for TCP processing in UNIX. Implicit processing priority increases from the left to the right. Device interrupts transfer data between the network interfaces and IP queues, and execute with the highest implicit priority (rightmost part of Figure 2.3). Only when these high-priority queues become empty does the kernel start processing work queued at the next-lowest priority. In this example, transport protocol processing would then occur (middle part of Figure 2.3). Finally, only when no work remains at either the device

level or the transport protocol level do user-level processes (leftmost part of Figure 2.3) receive the CPU.

One simple approach to provide idletime service would replace all resource schedulers in such a queue hierarchy with priority queues. Such a naïve approach is not suitable for general-purpose operating systems. Wide spread changes throughout the system are difficult to design and implement, require extensive testing, and can lead to API or service differences that cause application incompatibilities. Furthermore – as discussed in the next section in detail – priority queues alone are insufficient to isolate the foreground workload from the presence as idletime use.

A useful idletime mechanism will thus consist of a small number of localized modifications to key resource schedulers in a hierarchy. Such a mechanism can effectively control preemption cost, and thus isolate foreground performance.

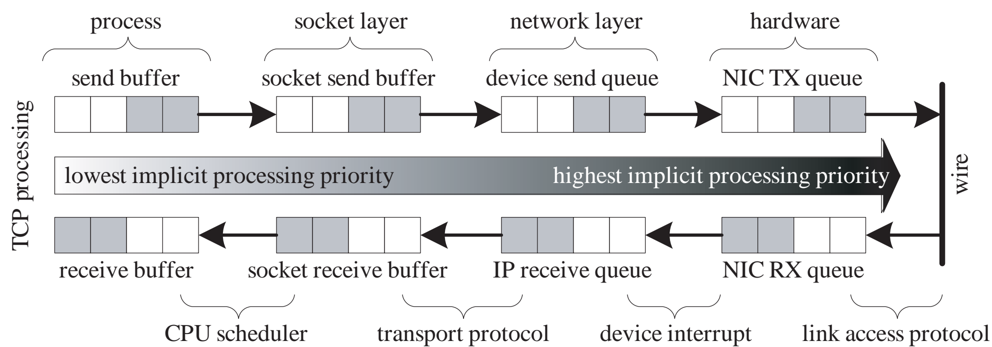


Figure 2.3. Queues involved in network communication over TCP, showing implicit processing priorities in UNIX.

2.3.3 Preemption Cost

The largest challenge faced to support idletime use of available resource capacity is preemption cost. Aborting one request and switching to another involves some amount of work and in consequence incurs a delay. For example, switching the CPU from one process to another requires a context switch (swap of the register set, flush of the cache, etc.) before execution continues. Each time a resource switches from idletime to regular use, this preemption cost incurs. Without idletime use, the resource would have been unused – ready to serve the new request immediately. The presence of idletime use thus delays foreground processing.

Figure 2.4 illustrates this preemption cost for a regular foreground request *R* by comparing scheduling without idletime use (top diagram) to scheduling in the presence of an idletime request *I* (lower diagram). In the lower diagram, idletime request *I* starts processing at t_1 and is still active when *R* arrives at t_2. The resource

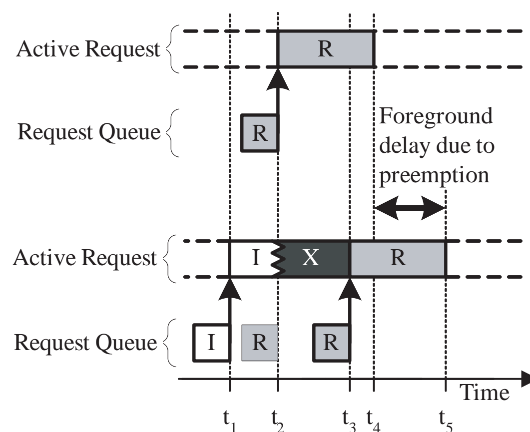


Figure 2.4. Preemption cost due to the presence of idletime use (bottom) compared to the non-idletime case (top).

immediately starts preempting *I*, which incurs a preemption cost (depicted by *X*). Thus, *R* cannot start processing until t_3, whereas it could start as early as t_2 in the absence of *I* (top diagram). It also finishes correspondingly later at t_5 instead of t_4.

Resources that frequently switch between different requests may have hardware support to minimize this overhead. One example is processors, which typically offer instructions to save and restore register sets to speed up context switches. Another example is customized I/O controllers for realtime systems [SPRUNT1988]. Such mechanisms can decrease foreground delay due to idletime scheduling, where they exist.

However, most other resources do not support preemption at all. For such resources, higher-priority requests must wait until an active lower-priority one finishes. Direct-memory-access (DMA) devices such as disk drives or network interfaces, which move data to and from memory without involving the CPU, fall into this category, because DMA transfers are usually non-preemptable. Idletime use of such a resource can significantly delay foreground processing, because processing times are usually much longer than preemption times. The delay while waiting for an active idletime request to finish will be much longer than waiting for the completion of a preemption operation.

The worst-case scenario is a workload with unlimited idletime load, where the arrival rate of foreground requests is slightly lower than the service rate of the resource. In

this case, the resource will start servicing a background request after each foreground request is finished, only to immediately preempt it when a new foreground request arrives. Each foreground request therefore incurs the preemption cost, and foreground delay and throughput are significantly affected.

Foreground delay is even higher for resources that do not support preemption. In that case, foreground requests must wait while the active idletime use finishes. Service times are usually much higher than preemption costs, leading to a reduction of foreground performance of up to 50%.

However, this worst-case scenario may not be frequent, and performance for more realistic workloads may still be acceptable. Most resource use tends to be bursty, such that a number of back-to-back foreground requests will preempt idletime use. In this case, the whole foreground burst only incurs a single preemption delay.

Preemption delays vary greatly for different resources. A CPU context switch typically takes a few microseconds [LAI1996], whereas a disk request may take dozens of milliseconds. It may be possible to disable idletime use of resources for which the aggregate preemption cost (i.e., the impact on regular performance) becomes too great, but continue to allow it for other resources. However, this approach may block background workloads that try to use idle capacity of the loaded resource. A successful idletime mechanism must carefully control preemption costs to minimize delays for foreground processing.

Additionally, a resource can eliminate the preemption cost in special cases where it knows the next occurrence of a foreground request. In such a case, it can stop idletime processing ahead of time, to push the preemption cost into the idle period. When the foreground request arrives, the resource is already idle and there is no delay. For certain periodic workloads, resources that require prior reservation, or jobs scheduled in advance (e.g., *cron*), such a scheme is possible.

This special case was one inspiration for the introduction of preemption intervals. They are periods of time following foreground requests during which no idletime work will be started, even if no further foreground request are enqueued. In some sense, the preemption interval acts as a "phantom" foreground request. To waiting background requests, the resource appears busy executing phantom foreground work during the preemption interval, delaying them until its expiration.

2.3.4 Tunability

With idletime scheduling, preemption costs and the corresponding reduction in foreground performance cannot be completely avoided. To support a wide variety of workloads (and user delay tolerances), the behavior of an effective idletime mechanism should be tunable.

One example is supporting different levels of aggressiveness in exploiting idle capacity. Higher levels allow higher idletime throughput, but may also incur higher

foreground delays. Supporting different levels for different resources allows further customization of the mechanism to particular usage scenarios.

2.3.5 Cache Pollution vs. Preload Effect

Another issue with idletime use may be cache pollution. Many hardware and software caches exist in a typical computer system to speed up operation. They replicate frequently/recently used data in faster storage space, to reduce retrieval latency on future use.

Caches are resources that benefit from using idle storage capacities. When caches themselves support idletime use, it can tag entries as foreground or background, and prevent new background entries from flushing existing foreground entries.

However, in the absence of idletime support for caches – such as with caches in existing hardware – idletime processing may create cache state that flushes entries created by regular requests. This may increase the delay of a future foreground request that then cannot take advantage of the cached data.

In these cases, it may be necessary to disable caching during idletime processing to prevent this effect [DOUGAN1999]. This will delay idletime use, but this performance decrease may be acceptable. Background tasks executing in idletime have by definition a lower importance (otherwise they should execute in the foreground), and their performance is of secondary concern.

However, in other scenarios, leaving caching enabled during idletime use may increase the performance of future regular requests. Some studies indicate that speculative preloads with useful data can speed up later processing [PIERCE1994][KWAK1999].

An idletime mechanism should default to disabling caches during idletime processing when possible, to minimize potential foreground delays. A selective override for specific caches and workloads is a useful feature that would allow exploitation of pre-load effects, where appropriate. An automated scheme that determines whether to enable or disable caching during idletime use could further improve performance.

2.3.6 Isolation of Side Effects

Isolation is the principle of hiding the side effects of idletime use from regular foreground processing. One obvious side effect is a decrease in foreground performance, on which priority queues and idletime preemption already concentrate. This section focuses on other, secondary side effects caused by the presence of idletime processing in the system.

The general idea is that the execution environment visible to foreground processes in the presence of idletime use must be identical to a system without idletime use. For example, using capacity in the regular file system to store idletime data (even if ample space is available) is problematic, because the files themselves would then be visible to foreground processes, and could interfere with regular processing. Another example

is an idletime process that binds to a specific network port and IP address – no foreground process can then bind to the same address/port pair.

Ideally, the system would use idle resources to provide an isolated, virtual execution environment, in which idletime tasks run. This is similar to FreeBSD *jails* [KAMP2000], Linux *vservers* [HUWIG2003] or *User Mode Linux* [DIKE2001], which restrict the set of system calls that super-user processes can execute, to improve system security. Other operating systems support similar virtual "sandbox" execution environments.

Isolation of idletime processing requires a set of capabilities from the sandbox environments that is different from their current focus on security. Sandboxes for supporting idletime processing must virtualize operating system state. Each idletime task executes in a separate sandbox, completely isolating their side effects from one another and from the regular foreground state. When idletime tasks finish successfully, the system can merge their respective state changes into the regular state visible to foreground processes. This atomic operation is similar to commit protocols used for databases, and may be even more challenging to realize.

Complete isolation requires the elimination of all shared state between sandboxes. Establishing this property in existing operating systems – without complete software emulation of all hardware, as in *VMware* [SUGERMAN2001] – will be extremely difficult and require widespread changes. Consequently, although complete isolation is

theoretically required to eliminate all possibilities for interference, it may be sufficient to virtualize a limited subset of operating system state to establish isolation in practical scenarios.

2.4 Summary

This chapter introduced the idea of productive use of idle resource capacity, and introduced how applications such as caches and prefetching schemes can benefit from such a service. It observed how the scheduling mechanisms in many current operating systems fail to provide this capability, how system architecture affects an idletime mechanism, and identified preemption cost as the main factor that delays regular foreground processing.

The next chapter will propose a general, resource-independent idletime scheduler that addresses these challenges, and describe its operation in terms of a more formal model.

3. Preliminary Work

Chapter 2 motivated the need for idletime use of available resource capacity. It can improve a wide variety of applications and services, including prefetching mechanisms, caches, disk and network service optimizations, and applications such as process or data migration systems. Chapter 4 will present the main contribution of this research effort, a generic, resource- and workload-independent idletime scheduler based on preemption intervals.

This chapter discusses preliminary work in idletime scheduling of network transmissions. It began the investigation into generic idletime scheduling that is the main contribution of this dissertation.

Section 3.1 presents and evaluates several application-level mechanisms [EGGERT1999] that enable idletime service in the *Apache* web server [APACHE1995]. It enables speculative push caching in the *LSAM Proxy Cache* [TOUCH1998]. *LSAM* uses background multicasts of related web pages, based on automatically selected interest groups, to load caches at natural network aggregation points. The proxy reduces server and network load, and increases client performance.

Experiments with the application-level mechanisms illustrate their effectiveness in establishing different service levels. However, the mechanisms do either not prevent degradation of foreground transmissions, or fail to utilize available capacity fully.

They do thus not support the required properties for idletime service described in Chapter 2.

Experience with the application-level mechanisms led to the investigation of kernel-level schedulers for idletime network scheduling that could address these issues [EGGERT2001A][EGGERT2001B]. Section 3.2 presents a preliminary kernel-level mechanism for idletime network scheduling, and evaluates its effectiveness experimentally. It finds that although this scheduler is more effective in isolating foreground traffic from the presence of background transmissions, it also fails to utilize significant amounts of available capacity for idletime use.

Furthermore, the mechanism's principle method of controlling background load is by dropping idletime packets. For network transmissions, which recover from packet losses at higher layers, a drop-based mechanism is acceptable. For a generic idletime scheduler, however, dropping part of the workload is not generally permissible. The generic idletime scheduler presented in the remainder of this dissertation uses preemption intervals as a general-purpose alternative to dropping partial workloads.

3.1 Application-Level Idletime Networking

The World-Wide Web is a typical example of a client/server system. In a web transaction, clients send requests to servers; servers process them and send corresponding responses back to the clients. Concurrent transactions with a server

compete for resources in the network and server and client end systems. Inside the network, messages contest for network bandwidth and with other messages flowing between the same end system pair and with other traffic present at the time. Inside the end systems, transactions compete for local resources while being processed. Servers implementing the process-per-request (or thread-per-request) model will allocate one process (or thread) to an incoming request.

The current web service model treats all transactions equivalently according to the Internet best-effort service [CLARK1988]. Neither the network nor the end systems typically prioritize traffic. However, there are cases where having multiple levels of service would be desirable. Not all transactions are equally important to the clients or to the server, and some applications need to treat them differently. One example is prefetching requests for web pages by proxies; such speculative requests should receive lower priority than user-initiated, non-speculative ones. Another simple example is a web site that wishes to offer better service to paying subscribers.

This section presents the design and implementation of three simple server-side, application-level mechanisms that approximate the idletime service model, in which background transactions never decrease the performance of concurrent foreground transactions. Slowing down the serving of background responses to make more resource capacity available to the average foreground response can approximate the ideal idletime service model. Experimental results show that the most effective application-level mechanism has an overhead on foreground performance of only 4-

17%. These results indicate that it is possible to provide effective background network service at the application-level.

3.1.1 Application-Level Idletime Mechanisms

This section presents the design and implementation of three server-side, application-level background processing mechanisms that approximate a service model with two classes: Regular foreground transactions, and preemptable, lower-priority background transactions. The design assumes a server that implements the process-per-request model, with pools of foreground processes and background processes. Because all mechanisms are server-side modifications, request transmission always occurs in the foreground. The mechanisms will only control processing and sending of the responses.

The key idea behind all three application-level idletime mechanisms is to slow down the background pool, thus making more resource capacity available to the average foreground process. The three mechanisms differ in how they slow down background processing. One assumption is that the operating system demultiplexes the request stream before it reaches the server. The server application consequently uses two sockets to accept foreground and background requests.

The first mechanism limits resource usage of background processes by limiting concurrency. It imposes an upper bound on the number of processes in the background pool. If all background processes are busy, additional incoming background

transactions are delayed (in the operating system) until a background process becomes available. The server does not enforce a bound on the foreground pool. Consequently, the average foreground transactions will experience less delay under an increasing background load, compared to a background transaction.

The size of the background pool is a parameter tunable by the administrator of the web server, based on the allowable overhead on foreground traffic. The experimental evaluation below used a five background servers. Fewer background servers would result in less background traffic, which would make it difficult to compare the overhead of the idletime mechanisms. Using many more than five would reduce the service difference between foreground and background traffic classes.

The second application-level idletime mechanism also limits the size of the background pool, but in addition also lowers the process priority of the background processes to the minimum. For CPU-bound servers, this approach should further reduce the impact of idletime transmission on concurrent foreground transactions, compared to the first mechanism.

The two prior mechanisms directly reduce only CPU usage. They can only indirectly control usage of network bandwidth and other resources. The third application-level idletime mechanism limits the aggregate network transmission rate of background processes by coordinating and scheduling their send operations. Background processes intentionally slow their transmission, monitoring and explicitly pacing their sending

rate by pausing while sending. Multiple background processes collaborate to split the bandwidth limit fairly. The rate limit is a parameter tunable by the administrator of the web server, based on the permissible overhead on foreground traffic. The experiments below use a rate limit of 1Mb/s. As with the first mechanism, a significantly lower limit would make comparisons of the idletime mechanisms more difficult, and a much greater limit would reduce the differences between the two service classes.

The third mechanism also limits the size of the background pool to five processes running at the lowest process priority. Note that for the third mechanism, limiting the background pool not necessary to enforce service differentiation – the send-rate limit establishes this. Limiting the background pool will simply control the send rate for each response: With only one background process, background responses occur at the full rate permitted by the rate limit (but only one at a time). With more than one background process, multiple background responses will transmit concurrently, each at a fraction of the rate limit. Lowering the process priority is also not strictly necessary, but because it is an extremely simple addition, it was added to the third mechanism.

One major issue with the third approach is that even if the network is underutilized, the background processes can never exceed the rate limit, because they have no means of detecting additional available network capacity. However, background transactions are not important by definition, so serving them at less-than-peak performance may be appropriate. The idletime scheduler presented in the remainder of this dissertation addresses this limitation.

None of the three background processing mechanisms rely on kernel-level or network-level support for service differentiation. However, if such support was available, they all could be easily modified to take advantage of such mechanisms.

3.1.2 Server Bottleneck Resource

An important step in designing an effective background processing mechanism is to locate the bottleneck resource of the system. Control of the bottleneck resource has primary influence on overall system behavior by granting or not granting the resource to processes. For example, in a CPU-bound system, a process that is not being granted the CPU cannot use other resources. Hence, the CPU scheduler controls system performance. In this scenario, network scheduling would have little effect on performance. A successful idletime mechanism will control the scheduling decisions of the bottleneck resource to optimize performance.

Any resource of a web server (CPU, physical memory, disk, network) may become the bottleneck, depending on the kind of workload it is experiencing. This section present a simple experimental investigation to determine the bottleneck resource in two web-serving scenarios: a web server connected to its clients by private, non-switched 10Mb/s and 100Mb/s Ethernet links. The server was monitored under a growing request load generated by an increasing number of clients, each of which made requests at a fixed rate of (at most) ten requests per second. The aggregate request load exceeded 1200 requests per second, which was more than enough to load the server fully.

The server machine was a 300MHz Pentium-II PC with 128MB of physical memory running FreeBSD 2.2.6. The kernel had been optimized for web serving by increasing the socket listen queue to 256 connections and increasing the *MAXUSERS* kernel parameter to 256. A modified *Apache* version 1.3b1 web server [APACHE1995] collected CPU, physical memory, page fault and physical disk I/O statistics. Server load was generated by a version of *WebSTONE* [TRENT1995] modified to gather more extensive per-request statistics. Each point in the graphs below averages data gathered during a five-minute period, in which several thousand requests were processed. No other traffic was present during the experiment. The network utilization hence simply corresponded to the amount of data transferred in a test period.

Both experiments issue requests over the standard *WebSTONE* page set, which is about 2MB in size and models a small, static web server. The entire file sets easily fit into the disk buffer cache of the server. Thus, the buffer cache will service repeated requests for the same file, and the disk subsystem was consequently mostly idle. Furthermore, all pages were static, i.e., no additional server-side processing (CGI scripts, database queries, etc.) was done.

3.1.2.1 10Mb/s Ethernet

The results for the 10Mb/s Ethernet case show that the server was network-bound during this experiment. The left graph of Figure 3.1 shows HTTP transaction throughput over the number of clients. Throughput quickly reached 7Mb/s and then

settled around that number. A single bulk TCP connection can achieve around 7.6Mb/s over the same link (measured with *Netperf* [HP1995]).

All other monitored resources were mostly idle: The server CPU utilization (right graph of Figure 3.1) was never higher than 25%. The experiments did also not fully utilize server memory; they caused no page faults. The disk subsystem was also idle; there were no physical (not served from the buffer cache) disk read operations. The disk output rate peaked at around 10 physical disk writes per five-minute test period, all of which were due to logging. The local file system can sustain several thousand physical disk writes per second at less than 25% CPU utilization, so the measured rate is not significant.

3.1.2.2 100Mb/s Ethernet

For 100Mb/s Ethernet, the server was CPU-bound. The right graph of Figure 3.1 shows that the server CPU utilization rose rapidly to around 95%. Network throughput stagnated at around 30Mb/s, (left graph of Figure 3.1) which is well below the

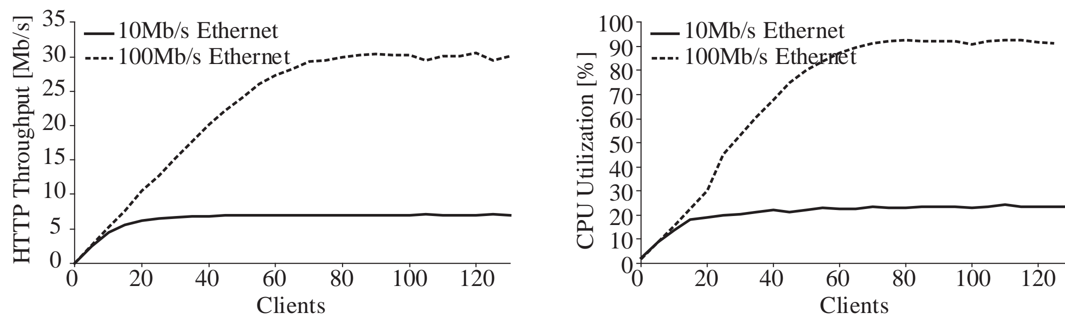


Figure 3.1. HTTP network throughput (left) and server CPU utilization (right) for both 10Mb/s and 100Mb/s Ethernet.

72.1Mb/s (measured with *Netperf* [HP1995]) that a single bulk TCP connection can achieve over the same link. The server was clearly not network-bound. The relatively low network throughput is likely to be an artifact of the *WebSTONE* benchmark, which only supports HTTP 1.0 and will thus open a new TCP connection for each transaction, causing significant CPU overhead.

As in the 10Mb/s case before, the experiment did not cause any page faults or disk input operations. The measured physical disk output rate never exceeded 50 writes per five-minute test run; as explained in Section 3.1.2.1, this rate is not significant.

3.1.3 Experimental Evaluation

All three idletime mechanisms described in Section 3.1.1 were implemented in *Apache* version 1.3b1 [APACHE1995]. The server machine was a 300MHz Pentium-II PC with 128MB of physical memory running FreeBSD 2.2.6. Two synchronized *WebSTONE* benchmark processes generated both foreground and background transactions [TRENT1995]. Each experiment kept foreground load at a fixed level while increasing background load over time. Increasing the background load will reduce foreground performance in a basic system. The three background processing mechanisms should reduce foreground performance degradation compared to the basic case.

Measuring the response time and size of each transaction allows quantifying the effect of background traffic on foreground load. Because replies of different sizes have different response times, the results below normalize response times against the fastest

measured time for the respective size for each network configuration. Normalized times are thus dimensionless. The best possible normalized response time is 1 (all responses took the minimum time). Because the experiments aggregate traffic from a number of clients, typical normalized times are 1-2 for light loads or 3-5 for heavier loads where foreground traffic has more self-interference.

To characterize the variability in measured traffic, the results below report median and quartiles of normalized foreground response times for all transactions measured during a five-minute test run (typically several thousand transactions). As background load rises, the median should rise and the quartiles spread, indicating higher interference and variability. The ideal background processing mechanism will minimize these effects, resulting in a flat, low foreground performance curve and a low inter-quartile gap.

3.1.3.1 10Mb/s Ethernet

The first two graphs show foreground response times in the basic case with no background traffic present. With one service class, median performance grew up to 40x worse (from 1.05 without background load to about 40) under light load (Figure 3.2, top row, left). It grew about 15x worse (from 2.8 to 42) under heavy load (Figure 3.2, top row, right). This case also saw a substantial increase in response time variation, as illustrated by the wide inter-quartile gap. Under heavy foreground load, there was substantial interference within the group of foreground connections. With no background traffic present, the median response time was 2-3x slower than under light

load. This indicates that background requests can substantially reduce median performance in an unmodified system.

The second row of Figure 3.2 shows the corresponding results from of the first idletime algorithm, where the server limited its background pool size to five. For both light and heavy foreground load, median performance only grew 5-6x worse. The simple idea of limiting the background pool resulted in a considerable improvement compared to the basic case. However, median performance decreased noticeably, and the variance in observed median performance was substantial, although smaller than in the basic case. This simple mechanism keeps median performance under 10x normal for half of all requests.

The second idletime algorithm also lowers the process priority of the background processes to the minimum in addition to keeping the pool size limited to five servers (Figure 3.2, third row). Median performance under light load was unchanged from the previous case (Figure 3.2, third row, left). Median performance under heavy load, however, was marginally better than during the previous experiment: 4x worse compared to 5x before (Figure 3.2, third row, right). Performance variance was also virtually identical to the previous experiment.

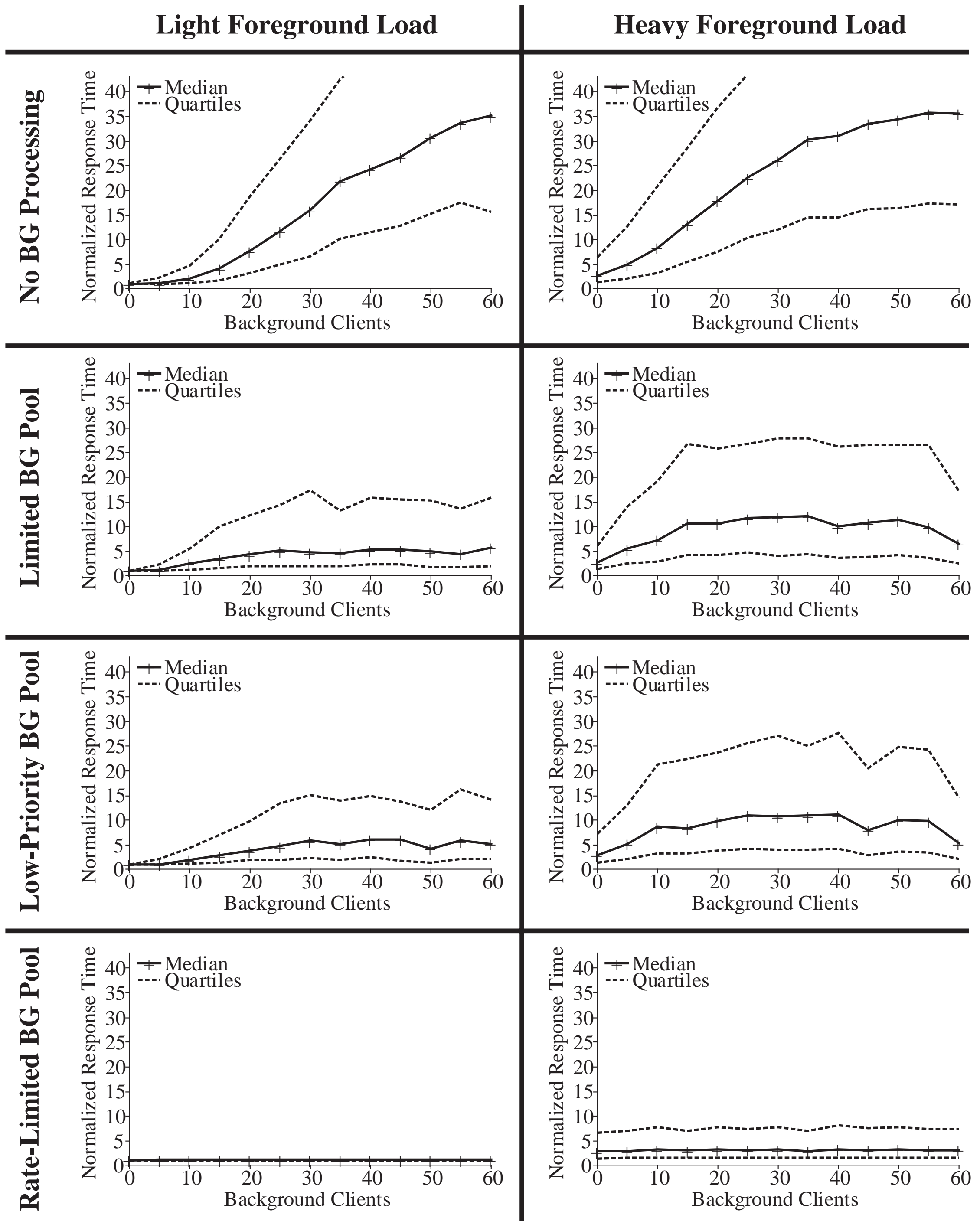


Figure 3.2. Normalized median foreground response times (with first and third quartiles) for the baseline case and three different application-level idletime mechanisms over 10Mb/s Ethernet; both under light and heavy foreground load.

As previously discussed, the CPU is not the bottleneck for the 10Mb/s Ethernet. Thus, even low-priority processes received enough CPU time to generate a substantial amount of network traffic. Process priorities are therefore not an adequate mechanism to establish different levels of service in this scenario. This result emphasizes the point that knowledge of the bottleneck resource is essential.

The third idletime mechanism rate-limited background sends (Figure 3.2, bottom row). It performed best, showing very low overhead and variance under both foreground loads. With light load, median performance grew by only 4%, and variance was extremely low (Figure 3.2, third row, left). Under heavy foreground load, median performance decreased by less than 18% (Figure 3.2, third row, right).

3.1.3.2 100Mb/s Ethernet

The performance expectations for the experiments over 100Mb/s Ethernet are different, because of the difference in bottleneck resources. As before, performance (both median and variance) decreased in the basic case with increasing background load: For light foreground load, it grew almost 10x worse: from 1.3 with no background load to about 11.6 (Figure 3.3, top row, left). For heavy load, it grew from 2.8 to almost 16: over 5x worse (Figure 3.3, top row, right). Variance in both cases was extremely high.

Again, substantial interference exists within the group of foreground connections alone. With no background load, median performance for heavy load is more than

twice as bad as for light load. Compared to the 10Mb/s case, the normalized response times here are about 50% lower than before. This is because in the network-bound 10Mb/s case, increases in response time are mostly due to packet loss and the resulting retransmissions. The 100Mb/s case has plenty of idle network capacity. Thus, increases in response time are mostly due to queuing inside the kernel.

Limiting the background pool improved both median performance and its variance under both sets of foreground load (Figure 3.3, second row). As for the 10Mb/s case, limiting the size of the background pool is an effective first step to establish different levels of service. Under light foreground load, median performance only grew worse by a factor or two (Figure 3.3, second row, left). It only increased by 40% under heavy load (Figure 3.3, second row, right). Again, this very simple mechanism could limit the impact background use has on concurrent foreground transactions.

The second idletime mechanism also lowered the priority of the background processes. This mechanism tries to address CPU-bound scenarios specifically. The experimental results indicate that the mechanism is not successful. Under both light and heavy background loads (Figure 3.3, third row), median performance is only marginally better than in the previous case, where the background servers ran at the same priority as the foreground ones (Figure 3.3, second row).

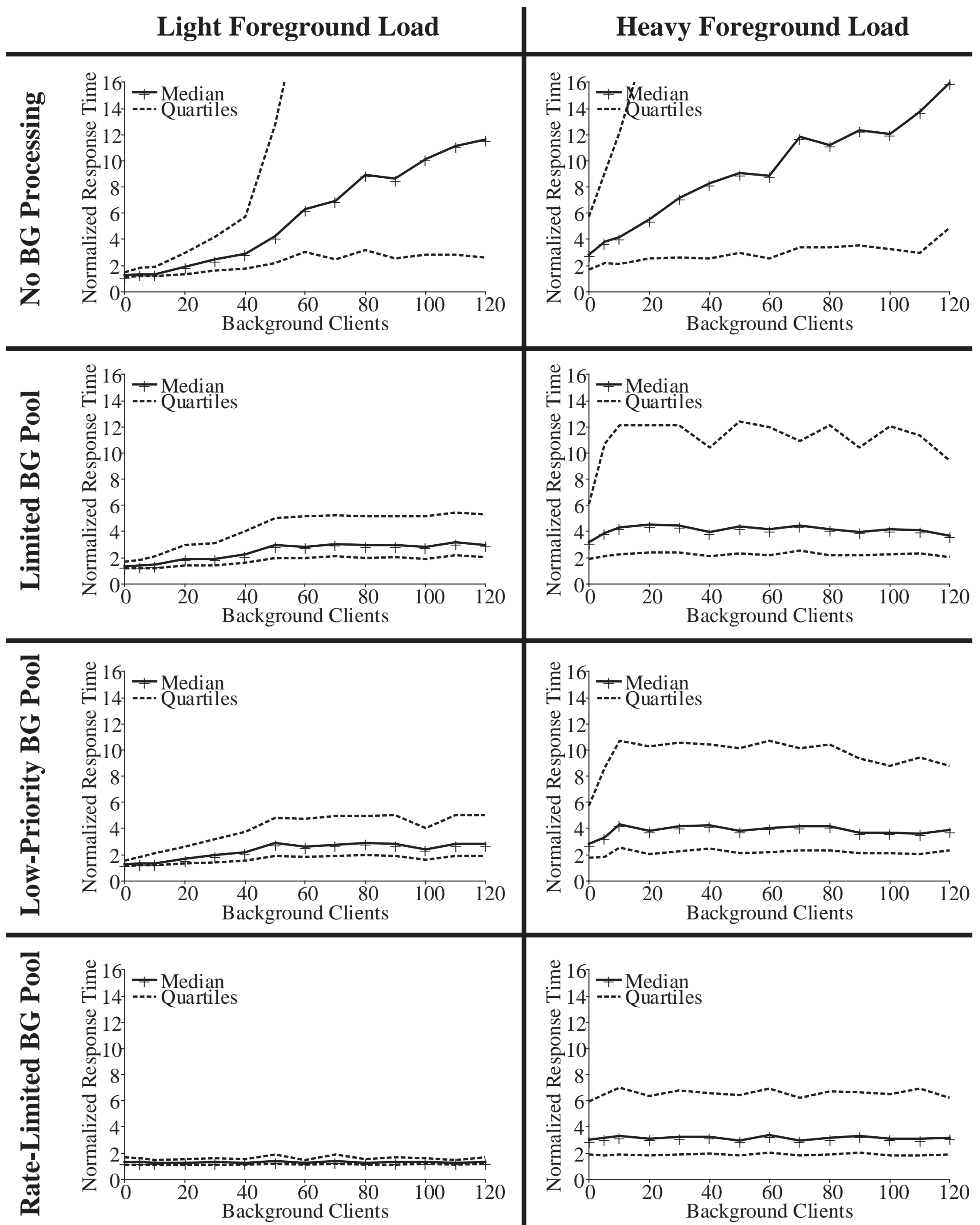


Figure 3.3. Normalized median foreground response times (with first and third quartiles) for the baseline case and three different application-level idletime mechanisms over 100Mb/s Ethernet; both under light and heavy foreground load.

One possible explanation for this lies in the nature of the 4.4BSD CPU scheduler [McKusick1996]. It lowers the priority of processes that have accumulated more CPU time than others have, and it raises the priority of blocked process. These two features of the scheduler counteract the mechanism's intention to use priorities to slow down background processes further.

Limiting the transmission rate of the background pool works best in this scenario, as it did before in the 10Mb/s case. Under light foreground load, median performance only degrades by about 6%, and the performance variance is extremely small (Figure 3.3, bottom row, left). Under heavy foreground load, median performance decreases by 11% (Figure 3.3, bottom row, left). This is only a moderately better than the first two algorithms. However, rate limiting significantly reduces response time variance, as indicated by the quartiles.

3.1.4 Discussion

An important first result of the experiment presented in Section 3.1.3 is that application-level mechanisms can provide substantially different levels of service. Even the very simple approach of limiting the background server pool works well in both experimental scenarios. It reduces median foreground response times to 5-10x the minimum for the 10Mb/s and 100Mb/s cases.

A surprising result is that the second mechanism, which also lowered the processing priority of a limited background pool, did not result in the expected improvement over

the first one, which just limited the pool size. Even in the CPU-bound scenario, where process priorities should be most useful, lowering processing priorities is ineffective. As described above, the BSD CPU scheduler eradicates the difference between high-priority and low-priority processes by rewarding I/O. On other systems, especially non-Unix systems, this may be different. However, because the mechanism improves median performance in minor ways in some cases and does not reduce it in the other ones, lowering the priority of the background pool can be a useful addition to other mechanism.

Of the three application-level idletime mechanisms, limiting the transmission rate of background processes performs best. In all experiments, median foreground performance decreased only by about 4-17% even under substantial background load.

One major issue with rate-limiting background transmissions is the inability of the mechanism to utilize more idle capacity than the fixed rate limit. If the operating system would provide accurate information about available capacity on short timescales, adapting the rate limit dynamically became possible. However, traditional operating systems do not provide such information, and applications have no means of passively detecting available network capacity.

Idletime mechanisms implemented inside the operating system can use internal kernel information to eliminate these issues. Kernel-level mechanisms may thus both provide better isolation of the foreground workload in the presence of idletime use, and utilize

more idle capacity for background work. The next section presents a preliminary idletime networking mechanism implemented as a kernel modification. Chapter 4 and the remainder of this dissertation then present a generic kernel-level idletime mechanism based on preemption intervals.

3.2 Kernel-Level Idletime Networking

Ideally, in a network with support for idletime use, background packet processing will only occur when resources would have been otherwise idle. Consequently, the presence of background traffic would be undetectable when observing foreground transmissions. In such a network, the background class can only use resources not already consumed by foreground transmissions. Starvation may occur: If foreground traffic saturates a link, background traffic will not receive service. Chapter 2 presented this distinction in service in detail.

Experimental results presented in the next section show that current operating systems are not effective in establishing such different service levels for network traffic. The event-driven, asynchronous nature of network stack processing interferes with attempts to use CPU-scheduler-based mechanisms as offered by current systems to control network send behavior.

Section 3.2.2 presents the design of a simple kernel-level mechanism to support idletime networking. It is a minimal extension of the current BSD network stack. The

modifications concentrate on the sender's network layer; transport protocols and socket API remain unchanged.

Section 3.2.3 presents an experimental evaluation of a prototype implementation of the new mechanism in the BSD network stack. The results suggest that it is effective in establishing idletime network service. Using the new mechanisms, foreground senders can achieve 97-99% of the baseline throughput, where no background use is present. It hence effectively isolates foreground transmissions from concurrent background traffic.

3.2.1 Effect of CPU Scheduling on Network Transmissions

One of the main tasks of an operating system is to control and schedule application access to system resources. To support a wide variety of applications, a general-purpose operating system employs simple and predictable schedulers, trying to provide fair service to all users of a resource.

Because the CPU has traditionally been the bottleneck resource in a system, its scheduler is typically more evolved compared to other resource schedulers. UNIX systems use a multilevel feedback queue [McKusick1996]. This scheduler favors interactive, bursty processes that do not fully utilize their allocated CPU quantum over compute-bound batch jobs, which do. It rewards bursty processes by increasing their priority, and punishes compute-bound ones by lowering theirs. Most I/O-bound

processes are bursty – they block during device operations – and thus achieve high CPU priorities.

Commonly, the CPU scheduler offers the user processes some degree of control over their priorities through the *nice* utility. Non-privileged processes may thus lower their priority from the default. (Increasing the priority is restricted to privileged processes).

Some POSIX-compliant systems [POSIX1993] offer three distinct priority classes for processes: realtime, regular, and idletime. A separate multilevel-feedback queue manages each class. Processes in higher classes preempt any lower class ones. Starvation of lower-class processes occurs when higher-class load increases to saturation.

Simple first-in-first-out (FIFO) schedulers organize access to most other resources. Though FIFOs by themselves do not assure fairness, they can do so in combination with a fairness-enforcing CPU scheduler. This is, because a process cannot issue any resource requests without a CPU to run on. FIFOs and other schedulers typically do not allow processes to influence their scheduling decisions.

Thus, current systems offer only two candidate mechanisms to implement idletime networking via process priorities: *nice* and POSIX scheduling. The remainder of this section will experimentally evaluate whether idletime networking approaches based on CPU schedulers are effective.

3.2.1.1 Experimental Setup

In these experiments, two copies of the same benchmark process run in parallel on a single host. The benchmark process is network-bound. It simply tries to send as much pre-generated random data to a second machine as possible. At the end of the experiment, the process reports the amount of data successfully sent. One of the two benchmark processes is the foreground sender, the other one the background sender.

Each benchmark process uses a fixed number (here: 3) of either TCP or UDP connections to send its traffic. A single TCP connection cannot overload an isolated network link due to TCP's congestion control algorithm. When sending with TCP, the benchmark blocks until one or more connections become writeable. It then writes a chunk of data on the writable descriptors and starts over. When using UDP, the benchmark process sends one message over each descriptor until the send call fails. This indicates that the outbound device queue is full, and the process allows it to drain by sleeping for 10ms before starting over.

One variable of the experiments is the intensity of the foreground sender. It controls the fraction of CPU time a benchmark process spends in the previously described sending loop. For a fraction of 0.1, for example, the process will only try to send traffic for 10% of its allocated CPU quantum. On BSD systems, the default quantum is 100ms, meaning the benchmark would generate send bursts of 10ms before sleeping for 90ms.

For each combination of transport protocol (TCP and UDP) and send intensity (full: intensity = 1, and light: intensity = 0.1), the experiment runs for 1 minute. The resulting throughput measurements are normalized against the maximum throughput a single foreground sender achieves in the absence of background traffic. The graphs in the remainder of this section show mean normalized throughputs with 95% confidence intervals over a series of 10 iterations.

The sending host that runs the two load-generators and the receiving host are identical workstations with 300MHz Pentium II processors running release 4.2 of the FreeBSD operating system. They are located on an isolated, switched, full-duplex 100Mbps Ethernet. This setup is network-bound; one machine can satiate the link with a CPU load of 55%.

The metric for effectiveness of the idletime networking mechanism is the throughput of the foreground sender in the presence of background traffic compared to the basic case where no background traffic is present. Better idletime mechanisms will yield higher foreground throughputs. An ideal idletime mechanism will allow the

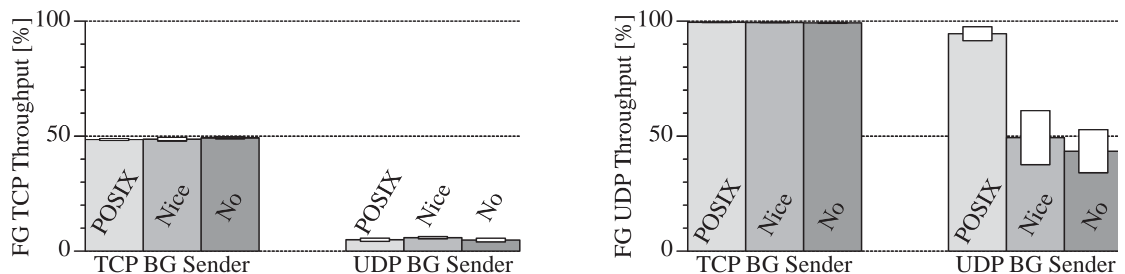


Figure 3.4. Normalized mean throughput with 95% confidence intervals of a foreground sender under unlimited load in the basic case (*No*) and with two CPU-based idletime networking mechanisms (*Nice* and *POSIX*), using TCP (left graph) and UDP (right graph).

foreground sender to reach 100% of the throughput it achieves in the absence of background traffic.

3.2.1.2 Experimental Evaluation

In the first experiment, the foreground sender sends TCP traffic at full intensity to the receiver. The left diagram in Figure 3.4 shows the measured and normalized throughput rates together with 95% confidence intervals (narrow white bars overlaying the wider gray bars).

With a background TCP or UDP sender, neither the POSIX nor the *nice* idletime mechanism can establish idletime network service that significantly improves on the basic case. The differences in throughput lie within the confidence intervals of the measured series. Furthermore, for a UDP background sender, this experiment demonstrates the worst-case scenario, because a background sender without rate-control can virtually shut down foreground transmissions. An effective idletime mechanism must adapt to this scenario, and protect foreground transmissions from idletime traffic. Both examined CPU-based schedulers fail to do so. The foreground sender only achieves around 5% of the possible throughput.

The right diagram in Figure 3.4 shows the case of a foreground UDP sender under background traffic created by TCP or UDP sources. With a TCP background sender, this case is the inverse of the worst-case scenario presented above: Here, the

foreground UDP sender monopolizes the link. Foreground UDP throughput is over 99% for all three cases, even the basic one.

If the background sender also uses UDP, the POSIX scheduler noticeably outperforms the nice one: 90% throughput versus 50%. Variations in throughput are also higher, as indicated by larger confidence intervals.

In the second experiment, the foreground sender is only active for 10% of its time quantum (= 10ms). Figure 3.5 shows the results for this case. When the foreground sender uses TCP to transmit its traffic bursts, the POSIX scheduler offers small foreground performance improvements of 5-10% over the basic case for both TCP and UDP background senders. Throughput does not increase with the *nice* idletime mechanism. With a UDP background sender, the POSIX scheduler increases performance 90% over both the basic case and the *nice* mechanism. Both the basic scheduler and *nice* are extremely ineffective in giving higher priority to bursty foreground traffic: it only achieves 1-3% throughput.

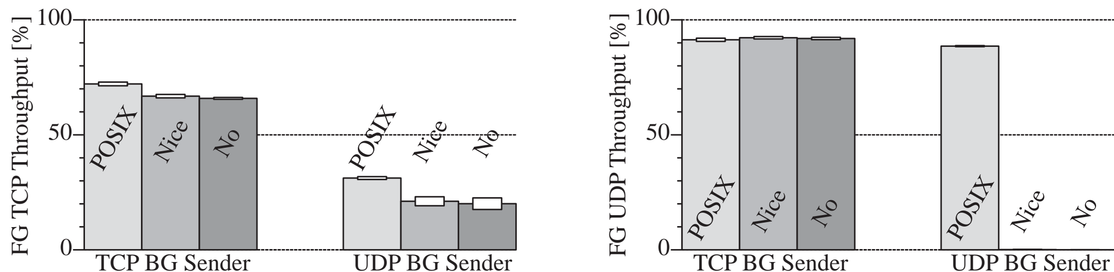


Figure 3.5. Normalized mean throughput with 95% confidence intervals of a foreground sender under bursty load in the basic case (*No*) and with two CPU-based idletime networking mechanisms (*Nice* and *POSIX*), using TCP (left graph) and UDP (right graph).

The experimental results presented in this section demonstrate that CPU-based mechanisms of current operating systems are not sufficient to implement idletime networking. Though scheduling background senders at POSIX idletime priority improves foreground performance in some scenarios (mainly for UDP senders), even then total foreground throughput only reaches 90% of the maximum. For other cases (e.g., foreground TCP senders), the POSIX mechanism only offers small performance improvement of 5-10%, increasing total throughput to 5-70%.

An effective mechanism to support idletime networking would achieve foreground send performances close to 100% in all of the above scenarios. It would also utilize significant portions of available idle capacity to transmit background traffic. The next section will present a kernel-based idletime scheduler for the BSD network stack that aims at improving upon the CPU-based mechanisms. Section 3.2.3 will evaluate the performance of the new scheduler.

3.2.2 Kernel-Level Idletime Mechanism

The key issue with the two CPU-scheduler-based candidate mechanisms for idletime networking lies in the event-driven nature of kernel network processing. Nearly all network processing – with the notable exception of UDP send operations – happens asynchronously with respect to application-level execution. Device interrupts trigger packet transmissions and receptions. Packet receptions trigger incoming transport protocol processing, which in turn may unblock processes waiting for data reception

on a socket. For TCP, packet receptions (and to a lesser degree, kernel timeouts) trigger packet transmissions.

In a sense, the network stack is an event-based system, where event priorities are lower layers have higher priorities (see Figure 3.6). As demonstrated by the experimental results in Section 3.2.1.2, the previously examined idletime mechanisms based on CPU scheduling have only very limited impact in such a system.

A second issue is the use of FIFOs for all kernel queues. The processing order of a FIFO queue is identical to the enqueue order, which may cause a queue's consumers to process earlier arriving background data before foreground data. This must not occur in a system supporting idletime networking.

3.2.2.1 Design Goals

The network stack is a complicated system, and many applications rely on its API (socket interface) and service semantics. Therefore, it is critical to avoid fundamental changes to the network stack. Additionally, much effort went into designing and fine-

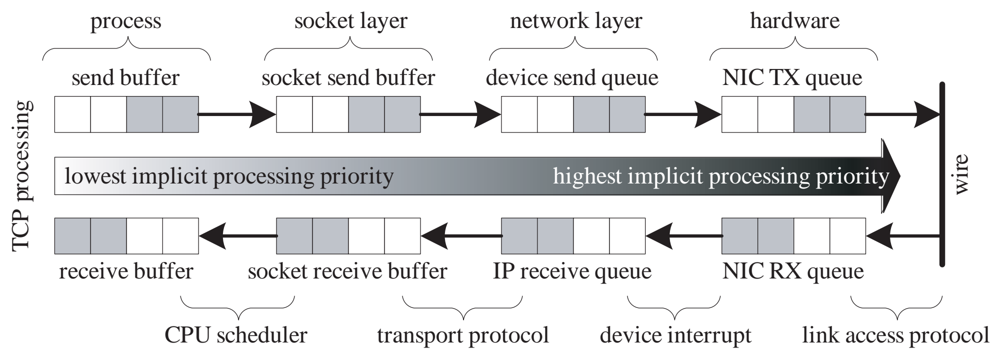


Figure 3.6. Network stack queuing and processing.

tuning the Internet’s transport protocols. Operating system extensions for idletime networking must not modify these transport protocols, to avoid incompatibilities with current standards.

It is also impractical to change all network drivers to support idletime networking, so hardware-dependent driver code must not change for idletime networking extensions. Note that part of the driver code is common to all devices of the same family; these routines could be safe to modify.

In addition, for end-to-end idletime networking, routers in the network must distinguish between foreground and background packets. This section assumes network support for idletime networking is available and the network handles packets according to their service marks. Chapter 5 discusses required network support in detail.

In summary, a design for OS extensions for idletime networking must be a simple extension of the current socket layer, must not modify the transport layer, and must not require changes to the hardware-dependent parts of device drivers – consequently, they must mainly extend the network layer.

3.2.2.2 Kernel-Level Design

One issue identified earlier in this section was the use of FIFOs for all queues in the network stack. To support idletime networking, two-level priority queues must replace most FIFOs in the network stack.

Part of the *KAME* IPv6/IPsec package [JINMEI1998] for BSD is the *ALTQ* framework [CHO1998] of alternate queuing disciplines. *ALTQ* replaces the outgoing standard FIFO queues of device drivers with configurable queuing disciplines, including priority queues. *ALTQ* already offers filtering of marked packets into a traffic classes. A small modification to the *ALTQ* priority queue that implements a drop preference for lower-priority when the queue is full establishes the desired behavior.

The service level of the network stack must not decrease for applications that are unaware of the presence of idletime networking service. Hence, the kernel must not send their packets in the background by default. Applications that are aware of the idletime service class may explicitly indicate the desire to transmit in the background to the kernel. The socket layer offers socket options to set user-configurable options on a per-descriptor basis. Thus, the only socket-layer change needed is a new socket option (*SO_BACKGROUND*) that indicates that the network stack should treat all traffic from or to a socket as background traffic.

Because of the event-based nature of the lower half of the kernel, drivers will transmit packets as soon as they enter their device queue: transmitter activation follows each

enqueue operation. Because the driver code executes at a higher implicit priority than the network layer (see Figure 3.6), the kernel typically transmits the packet before another one can be enqueued. Consequently, the network layer must verify if idletime packets may be send at a particular time (see below) before it enqueues them into the device queue.

The key idea behind the simple kernel-level mechanism for idletime network service is never to send background packets to a destination when a foreground sender is using the same outgoing interface. Instead, the network layer should drop these idletime packets, signaling an out of buffers (*ENOBUFS*) error condition. UDP senders must already be prepared to handle this error condition, because it occurs when the device queue fills up. They also must be prepared to handle data loss at the application. TCP senders will take the packet drop as an indication of congestion, lower the rate of the background sender, and recover the lost data through TCP retransmissions.

There are several possible methods to determine if an interface is in use by a foreground process before enqueuing a background packet into a device queue. The simplest approach is to check whether a foreground protocol control block (PCB) exists that uses the same outgoing interface. Although this simple approach is effective, it is also too restrictive: A single foreground TCP connection prevents the transmission of any background traffic– even while it is idle.

A more effective heuristic for detecting active foreground senders would not only check for the presence of a PCB, but also use additional means to determine if the PCB is an active user of the interface. For example, it could check if the socket associated with the PCB had any queued data in its send buffer. This would indicate an active sender. The prototype implementation evaluated in the next section uses this technique.

Active UDP senders are more difficult to identify. Unlike TCP, UDP does not buffer any data at the socket layer. This disables the heuristic described above. Furthermore, UDP write operations are non-blocking. They either succeed in enqueuing data into the device queue, or fail and return to the user process with an error. No kernel state exists that allows determining precisely whether a UDP sender is active at any given time.

The design for idletime networking thus uses the following heuristic to check for active UDP senders. For each UDP PCB, the network layer will check if the corresponding process is sleeping or not. A sleeping process indicates (paradoxically) an active UDP sender. This heuristic depends on the common structure of implementing UDP clients, which send until they fill the device queue or run out of data, then sleep to enforce a send rate limit. The implications of these assumptions are discussed in the next section.

3.2.2.3 Discussion

Several issues exist with the proposed design. First, the decision to enforce idletime networking at the network layer causes background packets to undergo socket and transport layer processing, only to be dropped when foreground senders use the same outgoing interface. Enforcing idletime networking at a higher layer would not incur this performance hit. This approach may prove problematic for more compute-intensive transport protocols, such as encrypted or tunneled flows.

A second issue is the per-packet PCB-lookup overhead to determine the active foreground senders for the same interface. The current implementation adds list of pointers to PCBs to each interface to limit the impact of this search. Other optimizations may further mitigate this overhead, but lie outside the scope of the initial prototype.

As discussed in the previous section, detecting active UDP senders at the network layer is difficult. The kernel can gain information about TCP connections and their corresponding processes from internal state. For UDP senders, no such state exists inside the kernel, because UDP senders manage it inside their respective applications. One possibility could be to extend UDP to queue data at socket send buffer, and to drain it as the interface queue empties. This would allow the TCP technique to check for active senders to extend to UDP senders. However, this approach significantly modifies the traditional UDP service semantics, and may fail to support existing applications.

3.2.3 Experimental Evaluation

This section repeats the experiments presented in Section 3.2.1.2 to evaluate the effectiveness of the idletime networking mechanism designed in the previous section. The experimental setup is unchanged from Section 3.2.1.1, except that the background senders are now using the new network idletime scheduler.

The following sections present two sets of experiment; one with unlimited background load, and one with a bursty background sender that only offers 10% load.

3.2.3.1 Unlimited Background Load

The left graph in Figure 3.7 shows how a fully loaded TCP foreground sender behaves in the presence of background load generated by TCP or UDP senders that use the kernel-level idletime network scheduler. In both cases, foreground throughput is over 99% of the baseline – a 50% improvement from the basic case.

The right graph of Figure 3.7 displays the result for a UDP foreground sender. Again, foreground throughput under unlimited background load reaches 97-99% for both

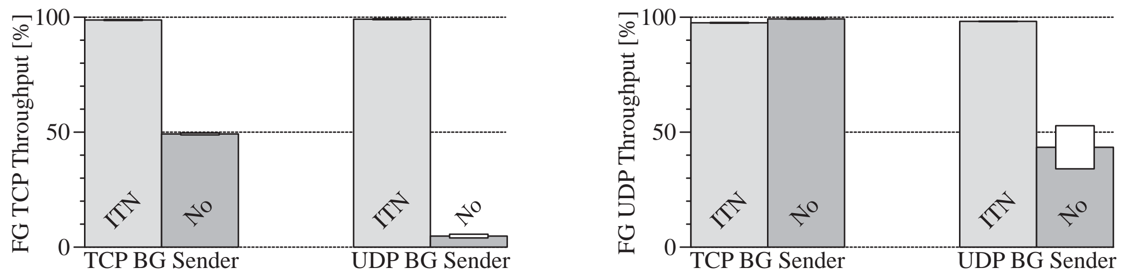


Figure 3.7. Normalized mean throughput with 95% confidence intervals of a foreground sender under unlimited load in the basic case (*No*) and with a kernel-level idletime mechanism (*ITN*), using TCP (left graph) and UDP (right graph).

UDP and TCP background traffic. With TCP background traffic, this is no improvement over the basic case. This is the worst-case scenario for TCP, due a high volume of congestion-unregulated UDP traffic. In fact, throughput is 1-2% lower, possibly due to processing overhead of the idletime mechanism. With a UDP background sender, performance increases by about 55% to 99%.

Figure 3.8 shows the raw, un-normalized foreground and background throughput numbers for the same experiment as a stacked bar graph. The dashed line indicates the baseline throughput that the foreground sender achieved when no idletime use is occurring. The graphs show foreground throughput as lightly shaded bars and background throughputs as darker shades of gray.

Without the kernel-level idletime scheduler (bars labeled *No* in Figure 3.8), foreground and background transmissions share the link fairly, each receiving 50% capacity. With the idletime scheduler (bars labeled *ITN* in Figure 3.8), the foreground sender receives full link capacity, and idletime use stalls.

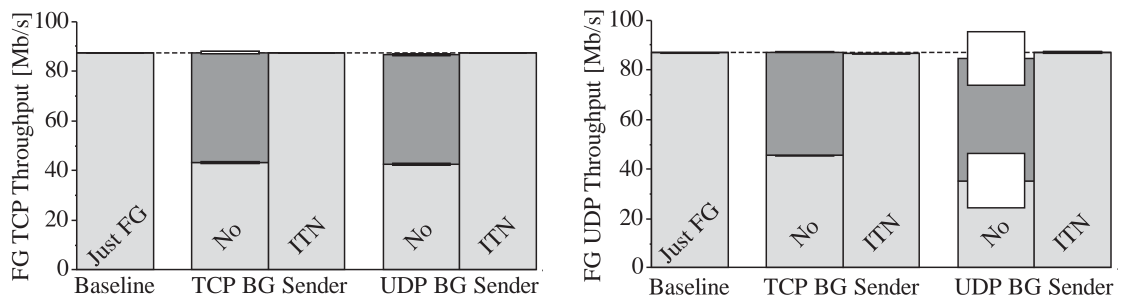


Figure 3.8. Mean foreground (light gray) and background (dark gray) throughputs with 95% confidence intervals under unlimited background load in the baseline case (*Just FG*), without idletime scheduling (*No*), and with a kernel-level idletime networking mechanism (*ITN*). Left graph shows TCP foreground throughputs, right graph UDP foreground throughputs.

In this first experiment with unlimited background load, the idletime network scheduler is effective in isolating foreground traffic from the presence of any background traffic in all four scenarios.

3.2.3.2 Limited Background Load

The next experiments look at the performance of a bursty foreground sender using the kernel-level idletime network scheduler. This background sender uses only 10% of its allocated CPU cycles to generate load. For a TCP foreground sender (left graph in Figure 3.9), the kernel-level idletime mechanism improves foreground throughput between 35- 99% baseline for both TCP and UDP background traffic.

For a UDP foreground sender (right graph in Figure 3.9), the idletime method also increases throughput to 99% for both TCP and UDP background traffic.

With a TCP background sender, this is a minor improvement of 5% over the basic case. Again, this experiment models the worst-case scenario for TCP, due to the presence of unlimited UDP traffic. For a background UDP sender, the performance

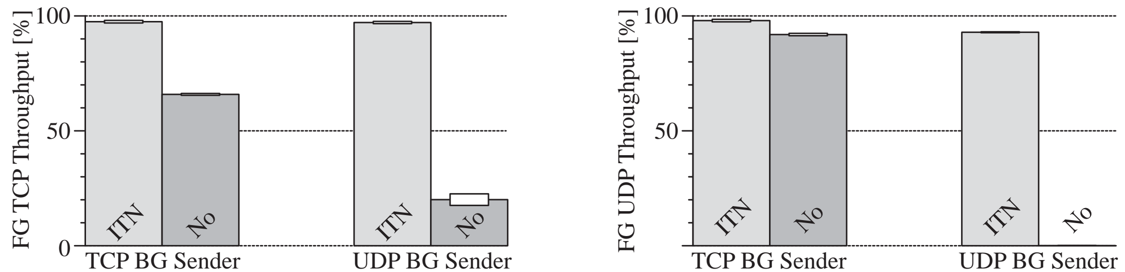


Figure 3.9. Normalized mean throughput with 95% confidence intervals of a foreground sender under bursty load in the basic case (*No*) and with a kernel-level idletime mechanism (*ITN*), using TCP (left graph) and UDP (right graph).

increase is around 90% – bursty foreground UDP traffic received very little service in the basic case, now its performance is close to the baseline maximum.

Figure 3.10 shows the raw, un-normalized foreground and background throughput numbers for the same experiment as a stacked bar graph, similar to Figure 3.9 for the previous experiment. As before, lightly shaded bars indicate foreground throughput, darker shades indicate background throughput. Figure 3.10 illustrates a key deficiency of the preliminary scheduler for idletime network service presented in this section. Although it is effective at isolating foreground performance from the presence of idletime transmissions, it fails to utilize significant amounts of available capacity.

With a foreground TCP sender, the idletime mechanism stalls a concurrent background TCP flow almost completely (Figure 3.10, left graph). Meanwhile, over 70% of the link capacity remains idle. An effective idletime mechanism should utilize this capacity for background traffic. A UDP background sender achieves approximately 15Mb/s throughput, but also leaves over 50% of the link capacity idle unutilized.

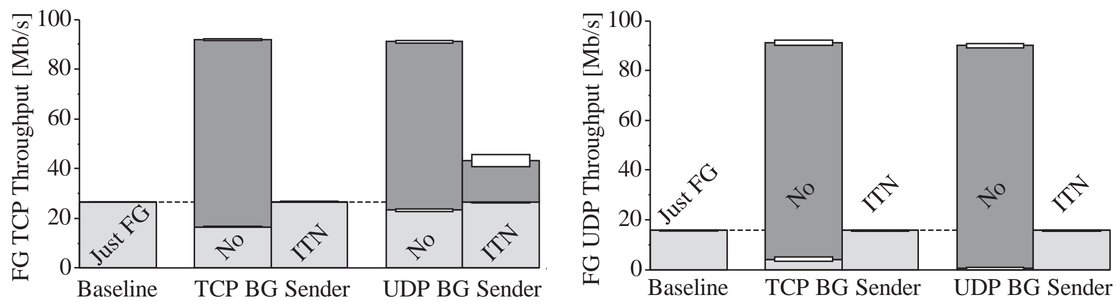


Figure 3.10. Mean foreground (light gray) and background (dark gray) throughputs with 95% confidence intervals under bursty background load in the baseline case (*Just FG*), without idletime scheduling (*No*), and with a kernel-level idletime networking mechanism (*ITN*). Left graph shows TCP foreground throughputs, right graph UDP foreground throughputs.

For a foreground UDP sender, the results are even worse. The idletime scheduler stalls both TCP and UDP background senders completely, and leaves over 80% of the link capacity unused (Figure 3.10, right graph).

These results show that although the idletime scheduler is effective in maintaining foreground throughputs at over 95% of the baseline performance, it fails to use significant amount of idle capacity for background transmissions. Due to the low idletime performance even in the presence of significant available capacity, the usefulness of the preliminary idletime network scheduler is questionable.

3.2.4 Discussion

The experimental results presented in Section 3.2.3 show that the preliminary kernel-level idletime network scheduler is effective in isolating foreground traffic from the presence of background traffic. In all investigated scenarios, foreground performance reaches 97-99% of the baseline case, in which no background traffic is present. This effectively isolates foreground transmissions from the presence of idletime use.

However, due to its inability to utilize available idle capacities for idletime use, the effectiveness and usefulness of the preliminary scheduler remain doubtful.

3.3 Summary

This chapter presented preliminary work in idletime scheduling for network transmissions. Section 3.1 evaluated several application-level mechanisms for idletime network service implemented in the *Apache* web server [APACHE1995]. Experiments with the application-level mechanisms illustrated their effectiveness in establishing different service levels. However, the mechanisms did either not prevent degradation of foreground transmissions, or failed to utilize significant amounts of available capacity.

Section 3.2 presented a preliminary kernel-level mechanism for idletime network scheduling, and evaluated its effectiveness experimentally. It found that although the kernel-level scheduler is more effective in isolating foreground traffic from the presence of background transmissions, it also failed to utilize significant amounts of available capacity for idletime use.

Another common limitation of both application- and kernel-level approaches is their focus on network scheduling. The application-level mechanism carefully determined the bottleneck resource of the web server, and used a specialized mechanism to control the application's use of the bottleneck resource. Though effective, this approach will not generalize to arbitrary resources.

Likewise, the kernel-level mechanism's principle method of controlling background load was dropping idletime packets. For network transmissions, which recover from

packet losses at higher layers, a drop-based mechanism was acceptable. However, a generic idletime scheduler cannot simply drop part of the workload.

The remainder of this thesis presents a generic, resource- and workload-independent idletime scheduler based on preemption intervals, which addresses the limitations of the preliminary mechanisms.

4. Idletime Scheduling with Preemption Intervals

Chapter 2 identified preemption costs as the key factor that controls foreground delay. In the worst case, every foreground request may require preemption of background work before receiving service, potentially resulting in significantly lower foreground performance.

This chapter proposes a scheduling mechanism for idletime use that minimizes impact on foreground performance for mixed workloads that incur preemptions. It establishes prioritization through priority queuing, and supports preemption when the underlying hardware and software can support it. The scheduler is also effective for resources that do not support preemption and require accordingly high context-switch costs. The first sections of this chapter will give an overview and discussion of the general idea. Later sections will formally define this mechanism and present a quantitative model to predict the general behavior of the mechanism. Chapter 5 describes a prototype idletime implementation for different resources in a real operating system and Chapter 6 evaluates the effectiveness of the prototype in a series of benchmarks.

The key feature of the proposed mechanism is limiting the aggregate preemption cost by introducing a *preemption interval*. A preemption interval is a time period following each serviced foreground request during which no background request will be started – the resource will remain idle, even when background requests are queued. The

preemption interval limits the cost to at most a single preemption per interval, when the preemption interval length is a multiple of the service time.

A similar idea exists in CPU scheduling. Instead of performing a context switch among processes (preemption) after each executed instruction, a CPU scheduler amortizes the cost of a context switch over a longer period (CPU quantum), during which a process can run without interruption. However, as previously discussed, the preemption interval for idletime scheduling only follows foreground requests, not idletime requests.

The basic operation of the proposed mechanism is as follows: the resource scheduler begins a preemption interval whenever an active foreground request finishes. While the preemption interval is active, the resource will not start servicing any idletime requests. It will service any queued foreground requests, however, and start a new preemption interval after each foreground request finishes. It will also immediately service newly arriving foreground requests. If no more foreground requests exist in the queue, the resource will remain idle until the preemption interval expires. Consequently, the resource will only start servicing idletime requests after the expiration of a full preemption interval in which no new foreground requests arrived.

The proposed idletime scheduler reduces foreground delays compared to a simple priority queue. Instead of immediately starting service for queued background requests whenever the last foreground request finishes, the resource remains idle. When a new

foreground request arrives at the resource, it can immediately receive service. With traditional priority queues, the new foreground request would have to wait for the resource to preempt or finish the active idletime request, decreasing performance.

Introducing artificial delays before idletime service relaxes the property of work conservation. Traditional schedulers are work conserving, because they do not allow the resource to remain idle while work is queued. The idletime scheduler relaxes this property by allowing the resource to remain idle for a limited amount of time before starting background work. Note that work conservation for regular foreground tasks remains in effect. Section 4.2 discusses these and other properties in detail.

The idletime scheduler purposefully delays idletime work to limit the impact of idletime use on regular foreground processing. It trades a reduction in idletime performance for the ability to limit the impact of idletime processing on regular foreground performance. The length of the preemption interval is a parameter that controls this tradeoff. With a longer preemption interval, the performance of idletime processing decreases, because each idletime request following foreground use incurs a long delay before it can start. Corresponding foreground performance will increase with a longer preemption interval, because the likelihood that the resource is busy serving idletime requests decreases. The length of the preemption interval thus allows tuning the idletime mechanism for particular resources and workloads.

The mechanism of the proposed scheduler is similar to *anticipatory disk scheduling* [IYER2001]. That work defines "deceptive idleness" that can lead to a reduction in performance for a work-conserving disk scheduler when multiple processes issue bursty disk requests. Anticipatory scheduling overcomes this issue by injecting short periods of idleness to stimulate the formation of request queues that improve the effectiveness of the *disksort* algorithm (see Section 5.3). The proposed idletime scheduler described in this work is a more general solution supporting arbitrary resources that specifically focuses on supporting different service levels. It was the result of an independent research effort that recognizes and counteracts a similar effect.

4.1 Formal Specification

The previous section gave an informal introduction of the proposed scheduling mechanism. This section describes the mechanism formally and in detail. It defines a model for resources and their operations, and discusses the specific properties required for idletime scheduling. Later sections present a simple mathematical analysis of the idletime scheduler, and derive theoretical estimates to compare against the experimental measurements of a real implementation presented in later chapters.

A formal definition requires a model of resource processing. For the purpose of this discussion, the following sections give a formal definition of a resource with its

axioms and operations. The model supports both spatially and temporally shared resources.

4.1.1 Definitions

A resource $S_t = \langle R, Q, A, c, f_c, \langle P, < \rangle, F, B, f_p \rangle$ at time $t \in \mathbb{N}$, where $\mathbb{N} = \{1, 2, 3, ...\}$, is a tuple with these elements:

Requests

R is the base set of all possible requests. $Q_t \subseteq R$ is the subset enqueued waiting for service at time t. $A_t \subseteq R$ is the subset that is being serviced by the resource at time t. (Without loss of generality, this discussion assumes that request identifiers are unique. Repeated identical requests receive unique identifiers, e.g., by including a nonce).

Capacities

$c \in \mathbb{N}$ is the capacity of the resource expressed as a natural number. $f_c : R \to \mathbb{N}$ is the required capacity of a specific request expressed as a natural number. Zero-capacity resources and requests are not useful.

Priorities

$\langle P, < \rangle$ is a partially ordered set of priorities. $F \subseteq P$ and $B \subseteq P$ denote sets of foreground and background priorities, respectively. $f_p : R \to P$

is the priority of a specific request. The priority of a request encodes its importance, f_p is an arbitrary function that assigns priorities to requests according to an arbitrary policy.

Required Capacity

$\overline{X} = \sum_{r \in X} f_c(x)$ is the required capacity for a set $X \subseteq R$ of requests.

4.1.2 Operations

A resource S_t supports the following operations, that when started at time $t \in \mathbb{N}$ commence at time $u \in \mathbb{N}$, where $u \geq t$. The duration $u - t$ indicates the service time of the operation.

Enqueue

$enqueue(r)$ adds a request $r \in R$ to the set of requests waiting for service such that $Q_u = Q_t \cup \{r\}$.

Service

$service(t)$ dequeues a request $r \in Q_t$ and moves it to the active set, such that $Q_u = Q_t - \{r\}$ and $A_u = A_t \cup \{r\}$. Which specific $r \in Q_t$ it serves depends on the queuing strategy of the resource. For idletime scheduling, the *dequeue* operation of the underlying priority queue will

always return the highest-priority request $r \in Q_t$, such that $\forall q \in Q_t : f_p(q) \le f_p(r)$.

Preempt

$preempt(r)$ moves a request $r \in A_t$ from the active set back to the queue such that $Q_u = Q_t \cup \{r\}$ and $A_u = A_t - \{r\}$.

Finish

$finish(r)$ removes an active request $r \in A_t$ from the service set such that $A_u = A_t - \{r\}$.

4.1.3 Axioms

A resource S_t at time $t \in \mathbb{N}$ must satisfy the following axioms:

Request uniqueness

No request can be both active and enqueued: $Q_t \cap A_t = \varnothing$.

Priority uniqueness

The sets of foreground and background priorities are disjoint: $F \cap B = \varnothing$.

Priority order

Foreground has priority over background: $\forall f \in F : \forall b \in B : b < f$.

Capacity limit

The active required capacity must never exceed the resource capacity: $\forall t \in \mathbb{N} : \overline{A}_t \le c$.

Satisfiability

All requests must be satisfiable: $\forall r \in R : f_c(r) \le c$.

Work conservation

Whenever a resource has capacity available to service an enqueued request $r \in Q_t$ at time $t \in \mathbb{N}$, it must immediately service r:

$$\left(\exists r \in Q_t : f_c(r) + \overline{A}_t \le c\right) \Rightarrow service(t).$$

(This is *strict* work conservation. Section 4.2.3 will replace this axiom with *weak* work conservation, to enable idletime scheduling with preemption intervals).

4.2 Idletime Properties

The basic model in the previous section can describe processing for many kinds of resources. Idletime use requires additional restrictions on resource operations in order to minimize the impact background work has on regular foreground processing.

Idletime use of available capacity depends on prioritization, preemptability, and isolation. This section will formally define these principles as properties of the resource model previously described. Informally, these principles are:

1. *Prioritization*: never process idletime requests while regular requests are waiting for service.
2. *Preemptability*: immediately preempt active idletime use to service incoming regular requests. Never preempt regular requests because of idletime use.
3. *Isolation*: the side effects of idletime use must remain hidden from foreground processing.

4.2.1 Prioritization

Prioritization is a function of work queue management. It reduces the impact of idletime processing by guaranteeing that regular foreground requests $r \in Q_t$ with $f_p(r) \in F$ will always receive service before any lower-priority, idletime requests $r \in Q_t$ with $f_p(r) \in B$.

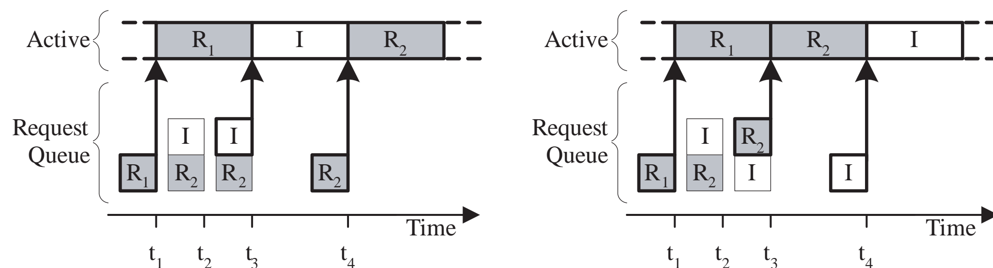


Figure 4.1. Temporally shared resource without (left) and with prioritization (right).

Prioritization

A resource S_t supports prioritization if and only if its $service(t)$ operation picks a new request $r \in Q_t$ to serve such that $p(r) = \max\left\{ f_p(q) \middle| q \in Q_t \right\}$.

4.2.1.1 Temporally Shared Resource

The left diagram in Figure 4.1 illustrates the operation of a FIFO scheduler for temporally shared resources that does not support prioritization. Before time t_1, the resource is idle. At t_1, regular request R_1 arrives and the resource immediately starts processing it, thus ending the idle period. At t_2, idletime request I and regular request R_2 arrive and are enqueued. At t_3, the processing of R_1 finishes.

The problematic scheduling decision occurs at t_3, when the scheduler picks idletime request I for processing instead of regular request R_2. Idletime processing for I delays regular processing (R_2 must wait until t_4 before receiving service), therefore violating the prioritization principle.

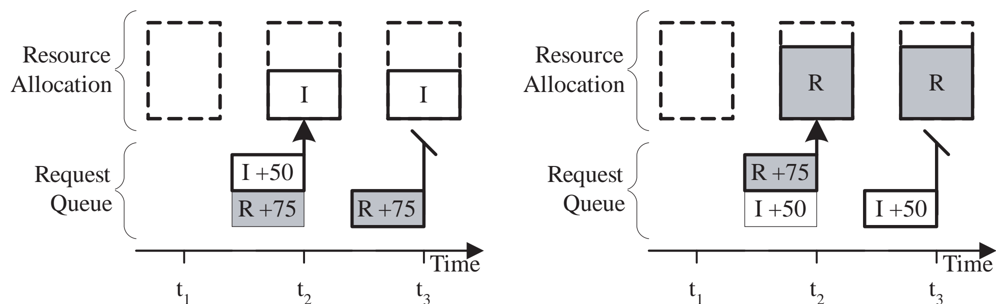


Figure 4.2. Spatially shared resource without (left) and with prioritization (right).

A scheduler with support for prioritization would have picked R_2 over I at t_3 instead (see Figure 4.1, right diagram).

4.2.1.2 Spatially Shared Resource

Prioritization is also a critical characteristic for idletime use of spatially shared resources. Figure 4.2 displays such a scenario for a spatially shared resource with 100 allocation units controlled by a scheduler without prioritization (left diagram). The resource is completely idle at t_1. At t_2, an idletime request I for 50 units and a regular request R for 75 units arrive at the resource. By allocating the capacity for the idletime request first, the scheduler causes the subsequent allocation of R at t_3 to fail due to insufficient capacity. This delays processing of the foreground request R until the user of I returns the block.

A spatial scheduler with support for prioritization (see Figure 4.2, right diagram) will schedule R before I. Even though the resource cannot service I at t_3 (again due to insufficient resources) and idletime use is therefore not possible, this scheduling decision does not delay foreground use.

4.2.2 Preemptability

Preemptability is a function of resource processing. It further reduces the performance impact idletime scheduling has on foreground processing by preempting ongoing idletime work whenever a new foreground request enters the queue.

Preemptability

> To support preemptability, a resource S_t must support prioritization and implement the following additional steps during its $enqueue(r)$ operation: whenever the idle capacity $\overline{I}_t = c - \overline{A}_t$ is $\overline{I}_t < f_c(r)$ and hence not sufficient to serve r, and there exists a subset $\exists P \subseteq A_t : \overline{P} \geq f_c(r)$ of lower-priority active requests such that $\forall s \in P : f_p(s) < f_p(r)$, the resource then preempts these requests $\forall s \in P : preempt(s)$ before immediately starting the $service(t)$ operation, which will then dequeue and service r due to prioritization.

Note that the resource may compose P as it sees fit. Different preemption strategies may result in different background service characteristics. For example, one strategy would choose the minimal P, such that it is the smallest subset of A_t whose preemption frees up enough capacity to serve r. This would maximize background throughput. Another strategy would pick the most recently started background requests in A_t, and as a result minimize wasted work.

Because the $preempt(s)$ operation re-enqueues the set of preempted background requests, they may receive service later, when the resource has enough idle capacity to start serving them.

4.2.2.1 Temporally Shared Resources

The left diagram in Figure 4.3 shows an example of a scheduler for a temporally shared resource. At t_1, it starts processing idletime request *I*. While it processes *I*, regular foreground request *R* arrives at t_2. However, the resource continues to process *I*, delaying execution of *R* until t_3, when *I* finishes.

The scheduler in this scenario violates the preemptability principle, because it does not immediately yield the resource to the newly arriving regular request *R* at t_2. The right diagram in Figure 4.3 shows how a scheduler with support for preemptability operates in the same scenario. At t_2, it preempts (or aborts) the active request *I* – incurring preemption cost *X* – and starts processing *R* instead at t_3. Thus, *R* receives service earlier than in the scenario without preemption, decreasing foreground delays. (For an ideal resource with zero-cost preemption, *R* would start immediately at t_2, due to the absence of *X*).

4.2.2.2 Spatially Shared Resources

The next example illustrates how a scheduler for a spatially shared resource supports preemptability, again for a resource with 100 allocation units. At time t_1 in the left

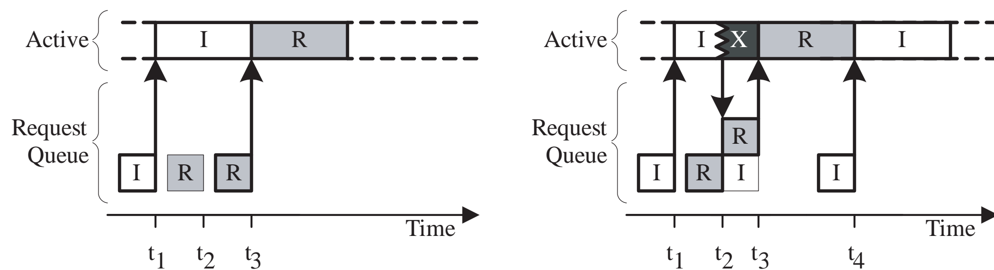


Figure 4.3. Temporally shared resource without (left) and with preemptability (right).

diagram of Figure 4.4, the resource is completely idle. At t_2, idletime request *I* for 50 units arrives, and the scheduler allocates the corresponding capacity. When a regular request *R* for 75 units arrives at t_3, it is declined due to lack of available capacity. This violates preemptability.

Instead of declining request *R*, a scheduler with preemptability support must reclaim (part of) the capacity allocated to idletime use (i.e., request *I*) whenever it has insufficient capacity for an incoming regular request. In the right diagram in Figure 4.4, the scheduler transparently reclaims 25 of the units allocated to idletime request *I*, so it can satisfy the regular request *R*.

4.2.3 Relaxed Work Conservation

The beginning of this chapter presented an informal description of the proposed idletime scheduler. The key idea is to relax the work conservation principle for idletime use. This allows a resource to remain idle for a limited time (preemption interval), even when idletime requests are waiting in the queue. Relaxing the work conservation principle can reduce foreground delays, because newly arriving

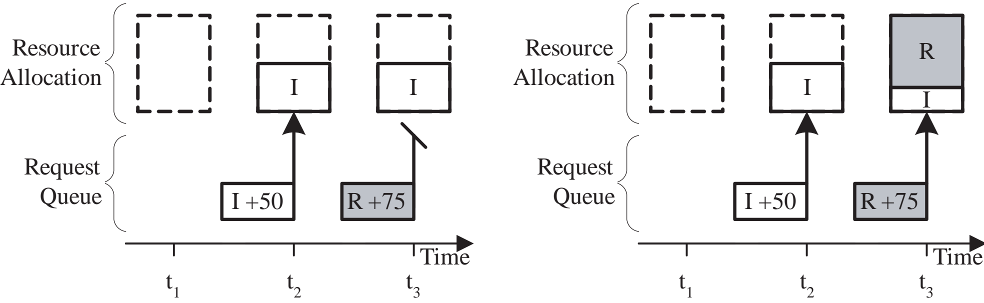


Figure 4.4. Spatially shared resource without (left) and with preemptability (right).

foreground requests receive immediate service during the duration of the preemption interval, without incurring preemption overheads. This section will define this property formally, and describe its operation in detail.

For reference, Section 4.1.3 defined work conservation – in the following discussion referred to as *strict* work conservation, to stress the difference – as follows:

Strict work conservation

> A resource S_t at time $t \in \mathbb{N}$ is work conserving, if and only if it services an enqueued request $r \in Q_t$ whenever it has sufficient capacity available: $\left(\forall r \in Q_t : f_c(r) + \overline{A}_t \leq c\right) \Rightarrow service(t)$.

The idea behind relaxed work conservation is to allow the resource to remain idle before servicing a queued request, even if capacity is available. The introduction of an arbitrary but limited idle duration prevents indefinite starvation under finite high-priority workloads:

Relaxed work conservation

> A resource S_t at time $t \in \mathbb{N}$ supports relaxed work conservation, if and only if it services, whenever it has sufficient capacity available, an enqueued request $r \in Q_t$ at some finite time $f_d : R \rightarrow \mathbb{N}$ in the future:

$\left(\forall r \in Q_t : f_c(r) + \overline{A}_t \le c\right) \Rightarrow service(t + f_d(r))$. (Note that if $f_d(r) = 0$, behavior degenerates into strict work conservation).

The key property of the proposed idletime mechanism is strict work conservation for foreground processing $\left(f_d(r) = 0\right)$, while requiring relaxed work conservation $\left(f_d(r) \ge 0\right)$ for background processing:

Idletime processing with preemption intervals

A resource S_t at time $t \in \mathbb{N}$ supports idletime use with preemption intervals, if and only if:

1. It is strictly work conserving for all enqueued foreground requests $r \in Q_t$ where $f_p(r) \in F$.
2. It requires relaxed work conservation for all enqueued idletime requests $r \in Q_t$ where $f_p(r) \in B$, using a function $f_d{}' : F \to \mathbb{N}$ that specifies the duration of the associated preemption interval for each foreground request.

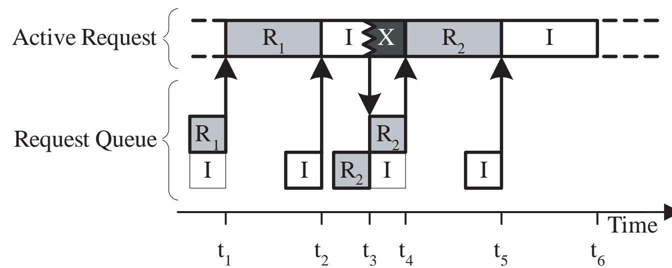


Figure 4.5. Scheduler with prioritization and preemptability, incurring preemption overhead.

The function $f_d' : F \to \mathbb{N}$ that specifies the length of the preemption interval following each foreground is deliberately simple to avoid complicating the model. Future extensions to this model could base the length of the preemption interval on other pieces of information.

Also, note that this definition of relaxed work conservation allows for idle periods between idletime requests. Section 5.1 discusses the possible variants of the idletime mechanism that can be derived as by-products of this definition.

Figure 4.5 gives another example of a scheduler that supports prioritization and preemptability, and the following sections will compare it to the idletime scheduler with relaxed work conservation. Note that even though the scenarios in the remainder of this section use a temporally shared resource, the mechanism works similarly for spatially shared resources.

In Figure 4.5, a background request I arrives at the idle resource at t_1, followed by foreground request R_1. The resource schedules R_1 first due to prioritization. At t_2, processing of R_1 finishes and processing of I starts. For the purpose of this and all following examples, context-switch costs associated with shifting the resource to the next request when the active one finishes are included in the respective processing times.

At t_3 – while the resource is still processing I – a new foreground request R_2 arrives. The resource preempts I (placing it back into the queue), and then starts processing R_2 at t_4. The preemption cost X in this example is $t_4 - t_3$. Finally, after R_2 finishes at t_5, the resource processes I, and becomes idle again at t_6. Note that preemption costs differ from context-switch costs previously described. Preemption costs occur while a request is underway, not when it has finished, and will always be shown separately.

Figure 4.6 shows the same scenario as Figure 4.5, but for a scheduler with a preemption interval. As before, a background request I and a foreground request R_1 arrive at the resource at time t_1, and the resource schedules R_1 first (prioritization). However, when R_1 finishes at t_2, the resource does not immediately schedule the queued idletime request I. Instead, it remains idle while waiting for the preemption interval started after R_1 to expire. Before the preemption interval expires, a new foreground request R_2 arrives at t_3 and immediately receives service. It does not incur a preemption delay, because the resource is idle at time t_3. When R_2 finishes at t_4, a new preemption interval starts. No new foreground requests arrive during that

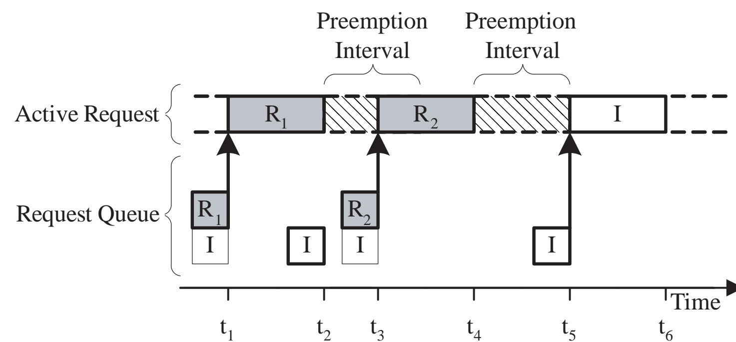


Figure 4.6. Idletime scheduler with prioritization and preemption interval.

preemption interval, and *I* finally receives service at t_5, when the preemption interval expires.

Note that a preemption interval will not prevent all preemptions. Instead, the duration of the preemption interval limits the impact idletime use has on foreground processing. With a short preemption interval, the system exploits idle capacities sooner, potentially increasing the amount of scheduled background work, but also increasing the preemption likelihood and in turn, the impact on foreground processing. Conversely, a longer preemption interval exploits less idle capacity for background use, but also reduces aggregate preemption costs. The evaluation in Chapter 6 explores the effect of different preemption interval lengths on foreground performance in several experimental setups.

The previous example (Figure 4.6) showed a scenario where the length of the preemption interval was sufficient to eliminate the preemption overhead. Figure 4.7 shows an example of a different scenario where preemption still occurs, even though the scheduler uses a preemption interval. This is because the length of the preemption

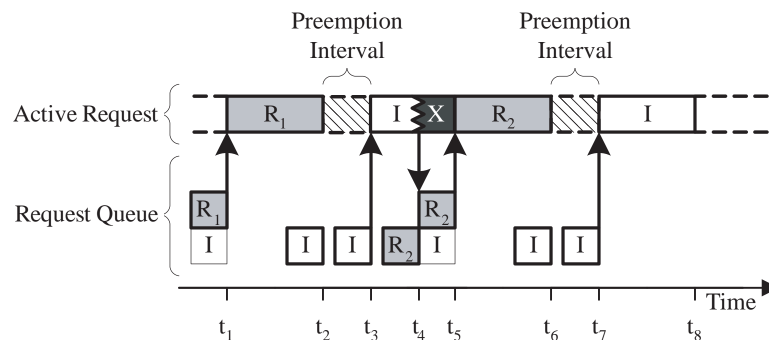


Figure 4.7. Idletime scheduler with prioritization and short preemption interval, incurring preemption overhead.

interval in this example is too short for the given workload. As before, a background request I and a foreground request R_1 arrive at t_1. R_1 receives service, finishes at t_2, and a preemption interval follows that ends at t_3. Now I receives service, but while it is active, the second foreground request R_2 arrives at t_4. The resource must now preempt I, move it back to the queue, and then start R_2 at t_5. This incurs a preemption delay $X = t_5 - t_4$. Only after R_2 finishes at t_6 and the associated preemption interval ends at t_7 does I receive service and finish successfully.

4.2.4 Isolation

The isolation principle states that all side effects of an idletime task must remain hidden during its execution. When it finishes successfully, the system may merge the changed idletime state into the regular foreground state in an atomic operation.

Prioritization and preemptability are sufficient to establish an idletime service class. However, without isolation, the state created as a side effect of idletime resource use can interfere with regular processing. One example of such interference could happen when idletime execution leaves a system data structure in an inconsistent state,

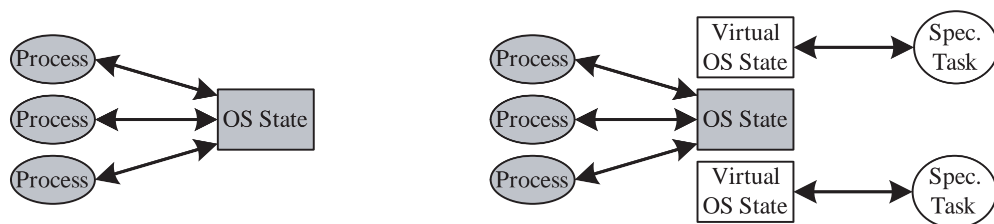


Figure 4.8. Shared operating system state (left) and virtualized operating system state (right).

because foreground use preempted it during modification of the structure.

The isolation property virtualizes the system state: instead of sharing state between all regular and idletime tasks, each idletime task executes with its own shadow copy of the state. Regular processes still access and share the master copy of the state, as before. Figure 4.8 shows the current sharing situation on the left side, and the virtualized operating system state on the right. Virtualized operating system state is similar to the concept of virtual memory, where each process executes in a separate address space.

Isolation is not a property of a single resource. Thus, the formal model previously defined for resource operation is insufficient to capture the desired properties. The remainder of this section will present a simple, general model of processing inside an operating system that can describe the isolation property formally.

State

Let S^* denote the set of all possible operating system states, where each $S \in S^*$ incorporates all variables and structures visible to processes.

Process

A process is a pair $P = \langle S_0, F \rangle$ where $S_0 \in S^*$ is the initial operating system state at its start, and $F = (f_1, \ldots, f_n)$ is a sequence of operations

where each operation $f_x \in F$ is a transformation $f_x : S^* \rightarrow S^*$ on the set of operating system states S^*. Let $\hat{f}_k = f_1 \circ ... \circ f_k$ denote the successive composition of the subsequence $\left(f_1,...,f_k\right)$, where $1 \le k \le n$.

Intermediary states

For a process $P = \left\langle S_0, F \right\rangle$, let $S_k = \hat{f}_k(S_0)$ be the intermediary state after k operations, where $1 \le k \le \left|F\right|$. The intermediary states reached during process execution of P form the sequence $\hat{S} = \left(S_0, \ldots, S_{|F|}\right)$.

Idletime process

An idletime process $\overline{P} = \left\langle \overline{S}_0, \overline{F} \right\rangle$ executes in an extended operating system state $\overline{S}^* \supseteq S^*$ with an extended start state $\overline{S}_0 \in \overline{S}^*$ and $\overline{F} = \left(\overline{f}_1, \ldots, \overline{f}_n\right)$ where $\overline{f}_x : \overline{S}^* \rightarrow \overline{S}^*$. The extended state $\overline{S}^* - S^*$ is invisible to regular processes (see the "isolation" definition).

Concurrent execution sequence

Given a regular foreground process $P = \left\langle S_0, F \right\rangle$ and an idletime process $\overline{P} = \left\langle \overline{S}_0, \overline{F} \right\rangle$, a concurrent execution sequence of operations is any sequence $\tilde{F} = \overline{F} \cup F$ that retains the relative pair-wise order of the

sequences $\bar{F}$ and F, such that for all pairs $\langle f_x, f_y \rangle \in F \times F$, there exists a corresponding pair of operations $\langle f_a, f_b \rangle \in \tilde{F} \times \tilde{F}$ such that $(f_x = f_a) \wedge (f_y = f_b) \wedge (x < y \Rightarrow a < b)$, and likewise for all pairs of $\bar{F}$. This naturally extends to multiple foreground and idletime processes through iteration on the concurrent sequence.

Isolation

Given a regular foreground process $P = \langle S_0, F \rangle$, an idletime process $\bar{P} = \langle \bar{S}_0, \bar{F} \rangle$ and a concurrent execution sequence $\tilde{F} = \bar{F} \cup F$, the side effects of idletime processing are isolated if and only if for each $f_x \in F$ there exists a $\tilde{f}_y \in \tilde{F}$ such that $f_x = \tilde{f}_y$ and $S_{x-1} = \bar{S}_{y-1} \cap \bar{S}^*$. In other words, before each foreground execution step in the concurrent sequence, those parts of the intermediate state that are not part of the extended idletime state must be identical to the intermediate state in a scenario without idletime processing.

Note that this model supports pre-executing a regular foreground operation speculatively using idletime. The speculative operation $\bar{f}_x \in \bar{F}$ is a duplicate of a regular one ($\exists f_y \in F : f_y = \bar{f}_x$) and $x < y$ in the concurrent sequence $\tilde{F} = \bar{F} \cup F$. In this case, f_y is speculatively executed earlier in the sequence (as $\bar{f}_x$), but the effects

of the execution are kept in the extended state. During execution of f_y, they are moved from the extended into the regular part of the state space. Section 8.3 discusses isolation techniques in more detail.

A starting idletime process has no associated virtual state until it performs write operations on operating system state. Whenever an idletime process is about to perform a write operation on a data item, the system atomically copies that data item (or a larger piece of state containing the item, such as a page), and then executes the write on the copied item. These shadow copies are invisible to regular foreground processing. Read operations access virtual state if it exists for a given data item, and regular operating system state otherwise. The system updates the shadow copies together with the master copy on regular use. Update conflicts with other shadow copies cause idletime tasks that depend on them to abort (or enter recovery, if supported).

The state of successfully finishing idletime tasks moves from the shadow copies into the master copy through an atomic operation. This commit operation also updates shadow copies belonging to other idletime tasks. It is a variant of traditional copy-on-

Figure 4.9. Foreground update and state propagation (left); idletime state commit at finish and propagation (right).

write schemes [POPEK1981][FITZGERALD1986][RASHID1988] because of the state merge operation required for finishing idletime tasks. Because data items can change in both virtual and regular state – when concurrent regular processing writes to the same data item – the system must detect these write conflicts, and abort (or restart) the idletime use.

Figure 4.9 gives an example of operations on virtual states. The diagram on the left shows how a foreground state change (1) by regular process P_1 results in immediate updates to the virtual states VS_1 and VS_2 belonging to idletime tasks S_1 and S_2, in steps (2) and (3), respectively.

The right diagram of Figure 4.9 shows how an idletime modification of its virtual state VS_1 by S_1 in step (1) is atomically committed back to the master state when it finishes (2). It consequently becomes visible to regular processes P_1 and P_2. Furthermore, the commit operation triggers an immediate update (3) to idletime state VS_2, as if a regular process had modified the master state. Note that if the VS_1 had conflicted with the master state – maybe because a foreground process had modified the same structures – the commit would fail.

4.3 Discussion

This section will discuss the implications of the mechanism described in the previous sections. The first part of this section discusses the effects of different preemption

interval lengths, and gives a simple heuristic to identify useful interval lengths. The second part of this section describes how an idletime mechanism using preemption intervals can support resource hierarchies where some of the schedulers do not support idletime use.

4.3.1 Preemption Interval Length

The length of the preemption interval allows tuning the idletime mechanism for particular resources and workloads. The impact of idletime processing on foreground performance will decrease with a longer preemption interval, because the likelihood that the resource is busy serving idletime requests decreases correspondingly.

The description of the idletime mechanism in this chapter has not yet addressed what the length of a preemption interval should be. A method for determining an appropriate length for the preemption interval, given particular workload properties (burstiness, inter-arrival rate, priority distribution, etc.) and resource characteristics (service time, preemption time, etc.), requires a queuing-theoretical analysis of the scheduler, which would depend on the traffic model more than the idletime mechanism.

This section will identify some simple heuristics for setting the length of preemption intervals for a resource. More refined schemes for setting – and possibly dynamically adapting – preemption interval lengths are an area for future research.

In the extreme case of an indefinite preemption interval, no idletime work ever receives service, and foreground performance will be identical to a system without idletime scheduling. In the other extreme – a zero-length preemption interval – foreground performance will be identical to a system that uses traditional priority queues.

The idletime scheduler minimizes foreground delays by limiting the number of required idletime preemptions to at most one per foreground burst. This amortizes idletime preemption cost over a burst of foreground requests. The defining characteristic of these bursts are inter-request gaps that are shorter than the preemption interval.

Figure 4.10 illustrates this behavior. Here, the preemption of *I* at t_2 delays the start of R_1 until t_3. After R_1 finishes, the resource enters the preemption interval $P(R_1)$. Because the $P(R_1)$ is longer than the inter-arrival time between R_1 and R_2, *I* remains blocked, and the resource remains idle and ready to serve R_2 immediately at t_4. If *I* had been allowed to receive service, another preemption delay would have delayed R_2.The

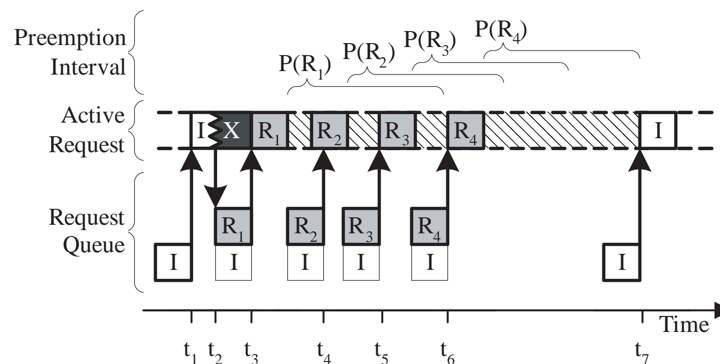


Figure 4.10. Bounded preemption cost through preemption intervals.

following foreground requests R_3 and R_4 are also part of the burst – their respective inter-arrival gaps are less than their respective preemption intervals – and consequently receive service without additional preemption delays. Only after R_4*'s* preemption interval $P(R_4)$ expires at t_7 does the resource allow idletime use, and starts servicing *I*.

Figure 4.11 shows an example where a long preemption interval prevents idletime service. At t_1, idletime request *I* is preempted, and the regular request R_1 receives service at t_2. When it finishes at t_3, its preemption interval $P(R_1)$ starts. Shortly before it expires, R_2 starts at t_4, and starts a new preemption interval $P(R_2)$ after it finishes, which further prevents *I* from receiving service. The same is true for R_3 and R_4, and their corresponding preemption intervals $P(R_3)$ and $P(R_4)$. Only after $P(R_4)$ expires at t_7 does *I* receive service.

A more realistic upper bound for the preemption interval length is the average foreground inter-arrival time. A preemption interval longer than the inter-arrival gaps will lead to a situation where idletime use remains disabled. To prevent this scenario,

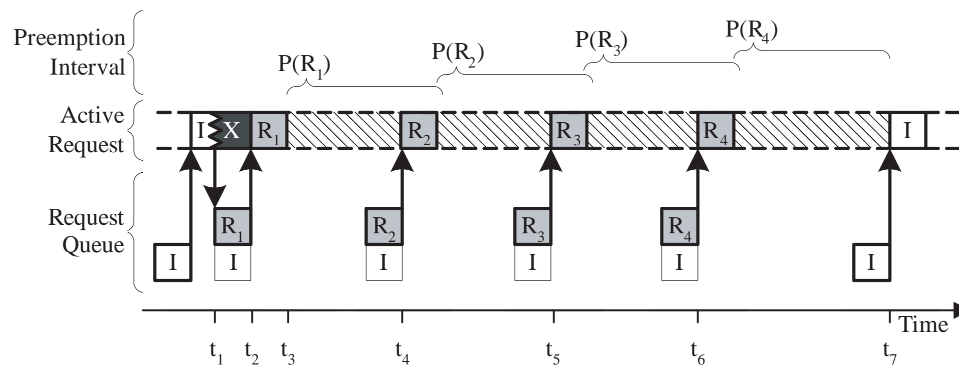


Figure 4.11. Long starvation of idletime processing caused by long preemption intervals.

and allow idletime requests to receive service, useful preemption interval lengths should be shorter than the foreground inter-arrival gaps.

However, preemption interval lengths cannot become arbitrarily short. The worst-case scenario for idletime scheduling is one-request foreground bursts with gaps longer than the preemption interval. In such a case, the mechanism is ineffective, and each foreground request will still incur the full preemption overhead. When a resource does not support preemption, foreground requests must wait on average half the service time for idletime work to finish before receiving service, causing up to 50% reduction in performance.

Figure 4.12 gives an example of a worst-case scenario. Here, the preemption intervals are shorter than the inter-arrival gaps, and *I* receives service after each of the preemption intervals $P(R_1)$ to $P(R_4)$ expire. Each time, the next new foreground request arriving while *I* is active incurs the preemption delay. Consequently, *I* does not successfully finish until t_{17}. This situation can only occur in two cases: very light foreground loads, or very short preemption intervals. Under very light foreground

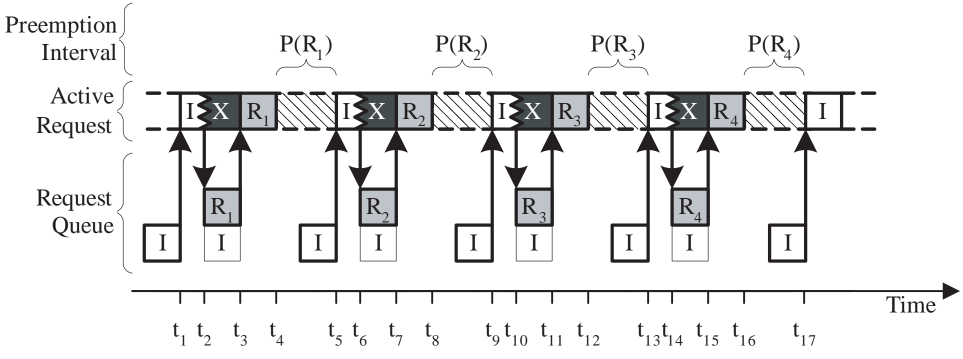


Figure 4.12. Idletime worst-case scenario.

loads, this behavior may be acceptable. When it is not, lengthening the preemption interval causes multiple spaced-out requests to form longer bursts, and thus increases amortization of preemption costs. However, a longer preemption interval also prevents utilizing the idle gaps for background use. In both cases, lengthening the preemption interval avoids this worst-case behavior.

Because the primary requirement for an effective idletime scheduler is the minimization of foreground interference (see Section 2.1), useful minimal preemption interval lengths would cause the formation of foreground bursts. This allows amortization of preemption costs. Longer preemption intervals will often result in longer bursts, and consequently reduce preemption overheads. The ideal preemption interval for a given workload is just long enough to cause the formation of foreground bursts that limit the preemption costs to within limits of a user-specified policy.

A strictly secondary objective is maximizing idletime work. This requires a preemption interval less than the average foreground inter-arrival time, which allows idletime requests a chance for service. These two rules can conflict. This occurs under high foreground loads, when the inverse of the inter-arrival rate approaches the service time. In such borderline cases, the first rule has priority and causes idletime starvation, protecting foreground performance.

4.3.2 Queue Hierarchies

Many approaches for service differentiation require extensive changes to both operating systems and applications, such as specification and enforcement of resource reservations. One benefit of the proposed idletime scheduler is that it can establish service differentiation through localized modifications.

Section 2.3.2 described how implicit scheduling decisions between the resources in a processing hierarchy could cause priority inversion and disable priority scheduling [LAMPSON1980]. One example is an operating system that processes queue events at lower hierarchy levels with higher priority. This effectively disables idletime use unless queues at all levels – and the implicit scheduling decisions between them – support priorities.

Another such example is a slower, non-prioritized producer queue feeding a faster, prioritized consumer queue. The left diagram of Figure 4.13 explains this effect in detail by showing the flow of a request stream through two chained queues over time.

Foreground (*F*) and background (*B*) requests arrive at a priority queue coming from a non-prioritized FIFO queue. Step (1) shows the start scenario. In step (2), the first foreground request flows to the priority queue, and enters its higher-priority queue at the top at step (3). Concurrently, the next (background) request *B* starts flowing to the priority queue. Because the priority queue is empty, it immediately passes the foreground request on to a downstream consumer in step (4). At the same time, the

idletime request *B* enters the lower-priority queue at the bottom, and the second foreground request *F* flows towards the priority queue.

In step (5), the priority queue is ready to schedule the next request. Because no foreground request is queued, it starts servicing idletime request *B*. When the second foreground request F arrives at the priority queue at time (6), it must either wait for B to finish (depicted in the diagram) or to be preempted. In either case, the second foreground request incurs a delay due to the presence of idletime processing, even though some queues support priorities.

The right diagram of Figure 4.13 shows the same scenario using an idletime queue as previously described in place of the simple priority queue. Steps (1) to (3) are identical

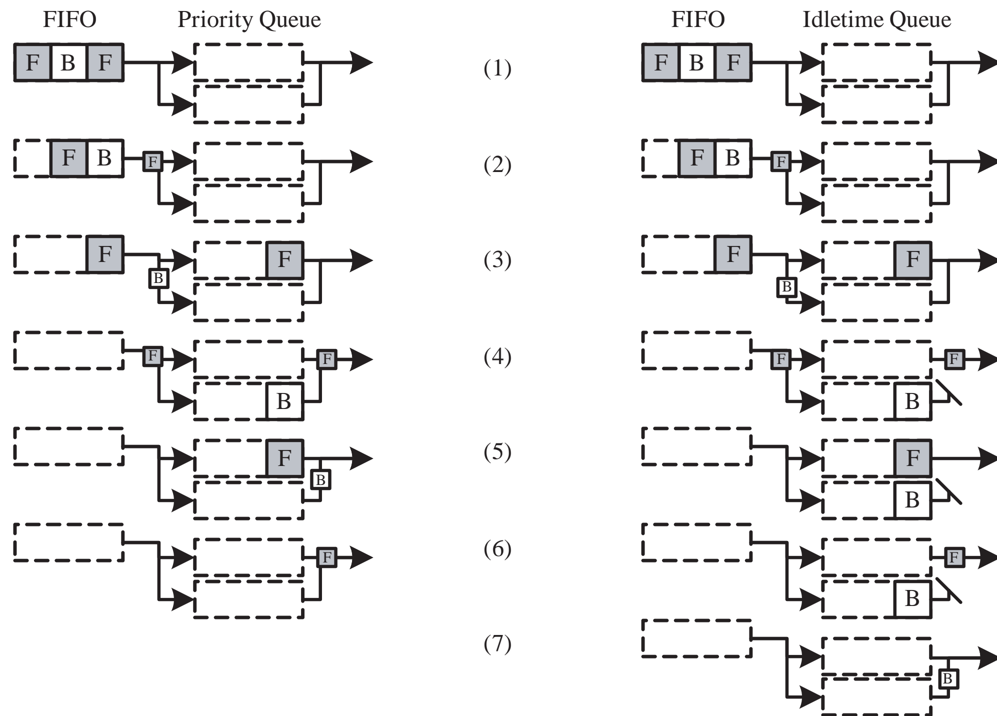


Figure 4.13. Operation of priority (left) and idletime schedulers (right) in a hierarchy.

to the previous scenario. In step (4), the idletime scheduler also starts servicing *F*. However, once *F* finishes, the scheduler enters a preemption interval, and will not start servicing idletime request *B*. In step (5), the second foreground request *F* arrives at the idletime scheduler while the preemption interval is still active. F can receive immediate service, and flows towards its consumer in step (6). Eventually, the preemption interval expires, and *B* receives services in step (7).

The example in Figure 4.13 illustrates how an idletime scheduler with a preemption interval decreases delays for foreground processing even in scenarios where some schedulers remain unmodified and do not support prioritization. It also demonstrates that this improvement comes at the expense of background performance; the idletime scheduler significantly delays the background request *I* until all foreground requests are finished, and the preemption interval has expired.

Note that the example assumed that the preemption interval was longer than the time difference between steps (4) and (5). A shorter preemption interval would expire before step (5), and the idletime queue would behave exactly like the simple priority queue on the left. Again, this illustrates how the length of the preemption interval controls the behavior of the mechanism.

4.4 Quantitative Analysis

This section will present a simple mathematical model that predicts the quantitative behavior of a resource managed by a scheduler supporting the idletime properties described in the previous sections.

The analysis presented here focuses on temporally shared resources, and quantifies the obtainable throughput. Figure 4.14 shows the configuration this section will analyze.

Two processes run concurrently, one issuing foreground requests, the other issuing background requests for the same resource managed by an idletime scheduler. The objective is to quantify the concurrent foreground and background performances achieved by the two processes.

Each process generates resource request at a certain rate, called the "intensity" of the process. It specifies the fraction of processing time a process uses to generate resource requests; for example, at 50% intensity a process spends half its cycles generating load and half its cycles doing other things (possibly sleeping).

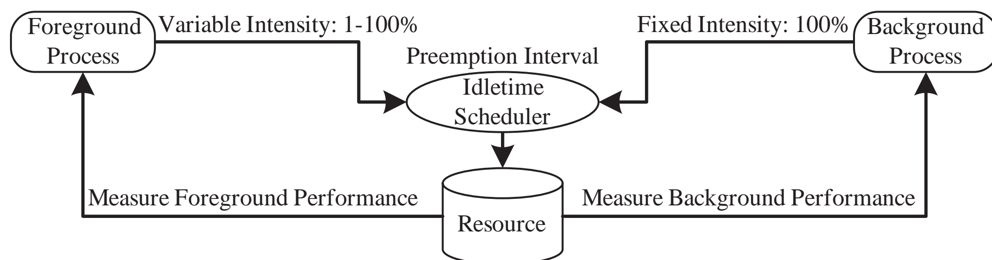


Figure 4.14. Analysis configuration.

The background process is greedy, and tries to consume as much resource capacity as possible: its intensity is therefore always 100%. This models the worst-case scenario for an idletime scheduler, because each foreground request could potentially incur a preemption delay.

The intensity of the foreground process is the first variable of the analysis and varies between 1-100%. This allows studying the impact of background processing as the foreground workload increases. The second variable of the analysis is the length of the preemption interval. This analysis focuses on throughput as a metric, and does not predict request latencies.

Goal

> Let $I = [0...1] \subset \mathbb{R}$ be the set of valid intensities, and $P \subseteq \mathbb{R}$ be the valid lengths of the preemption interval. Find two functions $f, b : I \times P \to \mathbb{R}$ that describe the achievable foreground and background performances at given intensities $i \in I$ and preemption interval lengths $p \in P$, respectively.

Several other constants are required for this analysis. One is the obtainable throughput at a given intensity $r : I \to \mathbb{R}$ when no idletime processing occurs. As a shorthand, let $r_{max} = r(1)$. Another is $t_{max} \in \mathbb{R}$, which specifies the maximal foreground request inter-arrival time in milliseconds. The former is resource-dependent; the latter is a system wide, resource-independent constant.

One of the key functions of the model is acting as an indicator for the correct operation of the idletime prototype implementation (see Chapter 5). Thus, the values for the constants $r : I \to \mathbb{R}$ (baseline performance without idletime presence) and $t_{max} \in \mathbb{R}$ (foreground inter-arrival time) were carefully chosen according to the measured or fixed characteristics of the benchmark system. This generates useful predictions that Chapter 6 will compare against experimental measurements of the prototype implementation, to determine whether the model is accurate.

Both the disk drive and network interface measured in Chapter 6 do not support preemption. Consequently, an active idletime request will delay a newly arriving foreground request until it finishes. For that reason, the quantitative model presented in the remainder of this section also assumes that the analyzed resource does not support preemption.

Another difference between the model derived in the remainder of this section and the previous definitions is due to implementation restrictions (see Chapter 5). The

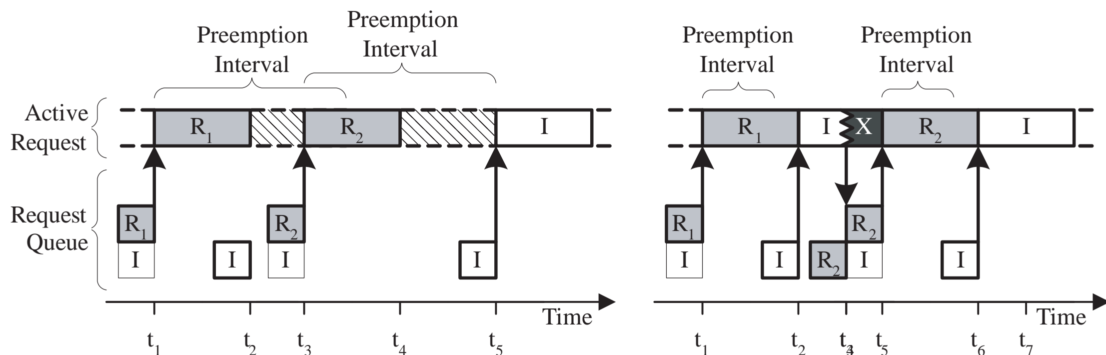


Figure 4.15. Effects of preemption interval length when preemption intervals start at the beginning of foreground requests.

prototype implementation begins each preemption interval at the *start* of the corresponding foreground request, and not when it finishes service. Figure 4.15 illustrates this difference. To allow for easier comparison of the prediction to the measured results, the quantitative model presented in the following sections also follows that behavior.

Note that this change does not affect any of the idletime properties previously discussed. The only difference is that for resources with fixed service times, a preemption interval less than the service time will not be effective. Such short preemption intervals will expire while their corresponding foreground requests are still active, as shown in the right diagram in Figure 4.15. In these cases, the idletime mechanism degenerates into a simple priority queue. For resources with varying service times for similar requests, the idletime scheduler will increase in effectiveness over a range of preemption interval lengths that corresponds to the distribution of service times.

Background Performance

> The term $a(i) = r_{max} - r(i)$ describes the maximum capacity the resource has available for background use when the foreground intensity is i, irrespective of the preemption interval p. The term $s(i) = (1-i)t_{max}$ describes the mean foreground inter-arrival time at intensity i. The preemption cost at intensity i is the inverse of the

service rate $r(i)$, based on the assumption that the resource does not support preemption. Consequently, with preemption intervals longer than the service time but less than t_{max}, the total time the resource is available for background use is:

$$\left(s(i) - p + \frac{1}{r(i)} \right) \quad \text{iff } \frac{1}{r(i)} \leq p \leq t_{max}$$

The fraction of achievable performance with preemption intervals in that region is:

$$\left(\frac{s(i) - p + \frac{1}{r(i)}}{s(i)} \right) \quad \text{iff } \frac{1}{r(i)} \leq p \leq t_{max}$$

The actual background throughput is consequently:

$$b(i,p) = \left(r_{max} - r(i) \right) \left(\frac{(1-i)t_{max} - p + \frac{1}{r(i)}}{(1-i)t_{max}} \right) \quad \text{iff } \frac{1}{r(i)} \leq p \leq t_{max}$$

With a preemption interval $p > t_{max}$, background requests become stalled, and background throughput becomes zero. The fraction of achievable throughput is therefore:

$$b(i,p) = 0 \quad \text{iff } p > t_{max}$$

If the preemption interval p is less than the service time, the idletime scheduler degenerates into a simple priority queue. Assuming that the resource does not support preemption, each foreground request is on average delayed by half a service time, and background throughput is approximately

$$b(i,p) = \left(r_{max} - \frac{r(i)}{2} \right) \quad \text{iff } p < \frac{1}{r(i)}$$

With all subterms replaced, the result is:

$$b(i,p) = \begin{cases} r_{max} - \dfrac{r(i)}{2} & \text{iff } p < \dfrac{1}{r(i)} \\ \left(r_{max} - r(i) \right) \left(\dfrac{(1-i)t_{max} - p + \dfrac{1}{r(i)}}{(1-i)t_{max}} \right) & \text{iff } \dfrac{1}{r(i)} \le p \le t_{max} \\ 0 & \text{iff } p > t_{max} \end{cases}$$

The right image in Figure 4.17 shows a graphic illustration of these three regions.

Foreground Performance

The term $r_{max} - b(i,p)$ denotes the resource capacity not utilized by background work at a given intensity and preemption interval. $r(i)$ is

the performance the foreground process achieves with no idletime use present. Foreground performance will be the minimum of the desired performance and the fraction idletime processing did not utilize: $\min\left(r_{max} - b(i,p), r(i)\right)$.

Furthermore, the presence of idletime use causes context switches, which may change the internal state of the resource and can therefore affect foreground performance. The factor $0 \le \alpha \le 1$ captures this effect, and is a resource-dependent constant. With all subterms replaced, the result is:

$$f(i,p) = \min\left(r_{max} - b(i,p), r(i)\right) \cdot \begin{cases} \alpha & \text{iff } b(i,p) > 0 \\ 1 & \text{otherwise} \end{cases}$$

The left image in Figure 4.17 shows a graphic illustration of these three regions.

4.4.1 Performance Expectations

The following sections use the simple quantitative model previously described to analyze the expected performance for two temporally shared resources with different characteristics: a disk drive and a network interface.

As discussed previously, the definitions for the two constants $r: I \rightarrow \mathbb{R}$ and $t_{max} \in \mathbb{R}$ are based on measured characteristics of the benchmark system the prototype implementation will be evaluated on (see Chapter 6). The maximum foreground inter-arrival time $t_{max} \in \mathbb{R}$ is identical in both scenarios in the remainder of this section: $t_{max} = 100\text{ms}$.

The two estimation functions $f, b: I \times P \rightarrow \mathbb{R}$ return absolute throughput numbers. For easier comparison, the graphs in the remainder of this section normalize them as $f_n, b_n: I \times P \rightarrow [0...1] \subseteq \mathbb{R}$. The foreground performance is normalized by $r: I \rightarrow \mathbb{R}$, whereas the background performance is normalized by the absolute maximum:

$$f_n(i,p) = \frac{f(i,p)}{r(i)} \text{ and } b_n(i,p) = \frac{b(i,p)}{\max_{i \in I}\left(r(i)\right)}$$

The predictors for foreground and background performance are functions of two variables (intensity and preemption interval), and plotting them produces a 3D surface. Comparing the shapes of these surfaces directly is difficult, because they do not reproduce well on paper. For this reason, the analysis will use 2D contour plots of the respective function.

Figure 4.16 shows how the surface plot of an example surface function (left graph) translates into a more readable contour plot (right graph). The shades of the contour plot indicate local performance; lighter shades of gray indicate better performance, darker shades worse performance. For each scenario, two contour plots will show foreground and background performance.

Figure 4.17 shows the foreground and background performance plots for an idealized idletime scheduler, and identifies the regions of interest. There are three major regions based on preemption interval lengths: the area where the preemption interval is less than the service time, the area where the preemption interval is longer than the foreground request inter-arrival time, and the area in between.

With a preemption interval lower than the resource service time, the scheduler is expected to be ineffective, because each interval will expire while its corresponding

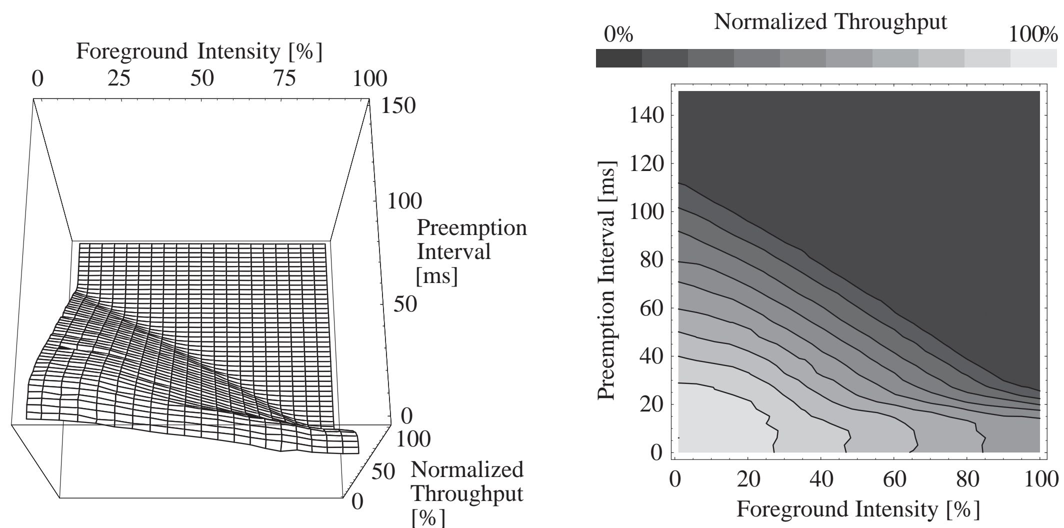


Figure 4.16. Surface plot of an example predictor function (left) and its corresponding contour plot (right).

foreground request is still active. Performance in this case will be identical to a system that uses traditional priority queue. Foreground performance in that case (left graph of Figure 4.17, bottom region) should be approximately 50%, because on average, each foreground request will incur a preemption of ongoing background work. Background performance in this case (right graph of Figure 4.17, bottom region) is close to 100% at low foreground intensities, and decreases to less than 50% at maximum foreground intensity.

In the middle regions of Figure 4.17, where the preemption interval length lies between the service time and the foreground inter-arrival time, the idletime scheduler becomes effective and maintains high foreground performance (light shades of gray). Background performance decreases with increasing foreground intensity due to more frequent preemption intervals, which decrease the amount of idletime capacity usable for background work. It also decreases with increasing preemption lengths, because

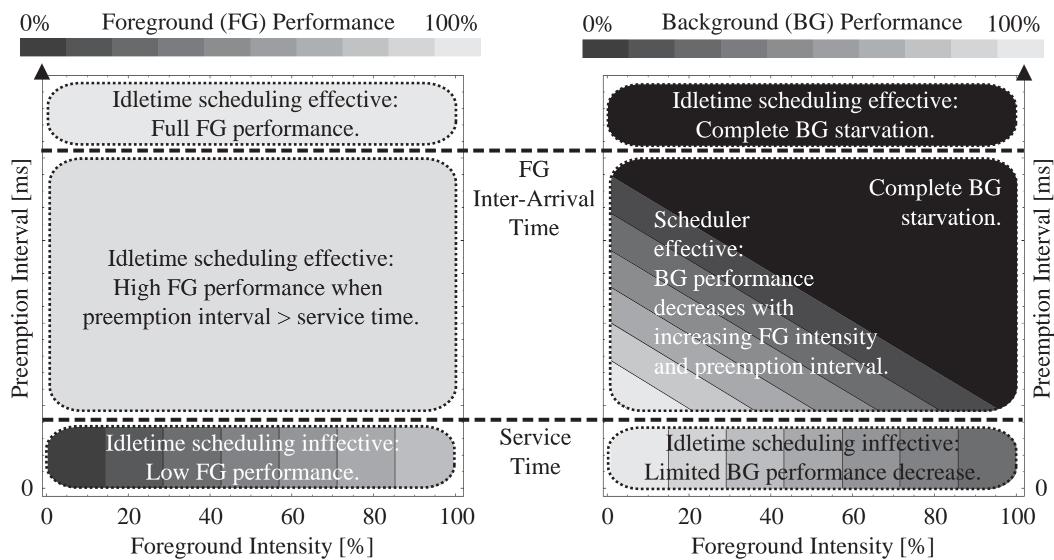


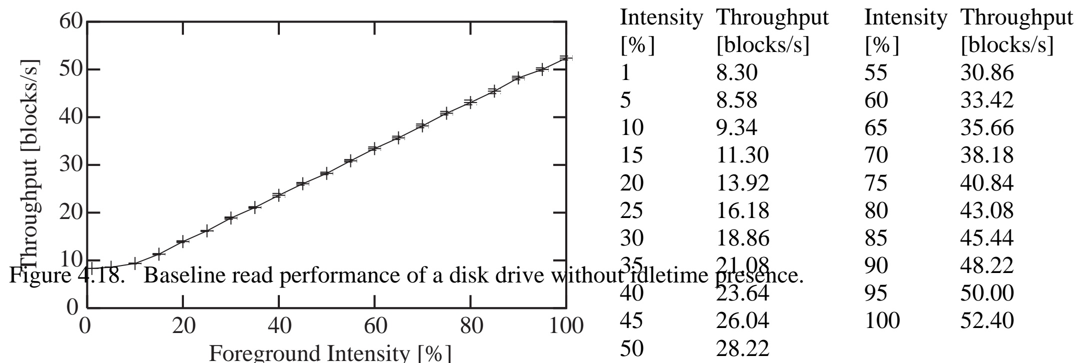
Figure 4.17. Overview of the expected performance behavior for an idletime scheduler.

longer preemption intervals at fixed arrival intensity also decrease the amount of idletime capacity available for background use.

Finally, in the third region in Figure 4.17, the preemption intervals are longer than the foreground inter-arrival rate. Here, idletime use completely stalls, because even at lowest foreground intensities, the inter-arrival time is always shorter than the preemption time, and preemption intervals never expire. Consequently, foreground requests achieve full performance.

4.4.2 Disk Drive

In the first scenario, the two load-generating processes issue *read* requests on the same file system. Each request reads a single disk block (512 bytes) at a random location on the drive. The baseline performance curve $r: I \rightarrow \mathbb{R}$ is based on measurements of a Western Digital Caviar AC28200 disk drive, shown in Figure 4.18. This drive has a measured mean service time of approximately 19ms, which conforms to the manufacturer's specifications of 15ms maximum mean seek time plus 5ms mean latency. The performance estimates presented in this section correspond to the experimental setup evaluated in Section 6.1.1 and will be compared against them in



Intensity [%]	Throughput [blocks/s]	Intensity [%]	Throughput [blocks/s]
1	8.30	55	30.86
5	8.58	60	33.42
10	9.34	65	35.66
15	11.30	70	38.18
20	13.92	75	40.84
25	16.18	80	43.08
30	18.86	85	45.44
35	21.08	90	48.22
40	23.64	95	50.00
45	26.04	100	52.40
50	28.22		

Figure 4.18. Baseline read performance of a disk drive without idletime presence.

Section 7.1.1.2.

Figure 4.19 shows the contour plots of the predicted foreground (left) and background (right) performance. They exhibit the same major patterns shown in the overview in Figure 4.17.

Looking at Figure 4.19 in detail, idletime use with a preemption interval shorter than the 19ms service time is ineffective, as expected. Foreground performance (left graph) in this case is approximately 50% of the base, while background processing (right graph) achieves between 50-90%, depending on foreground intensity.

With a preemption interval of more than the service time of 19ms, the idletime mechanism becomes effective. Foreground performance is over 90% (light areas in left graph) while background traffic still receives some service (lighter shades of gray

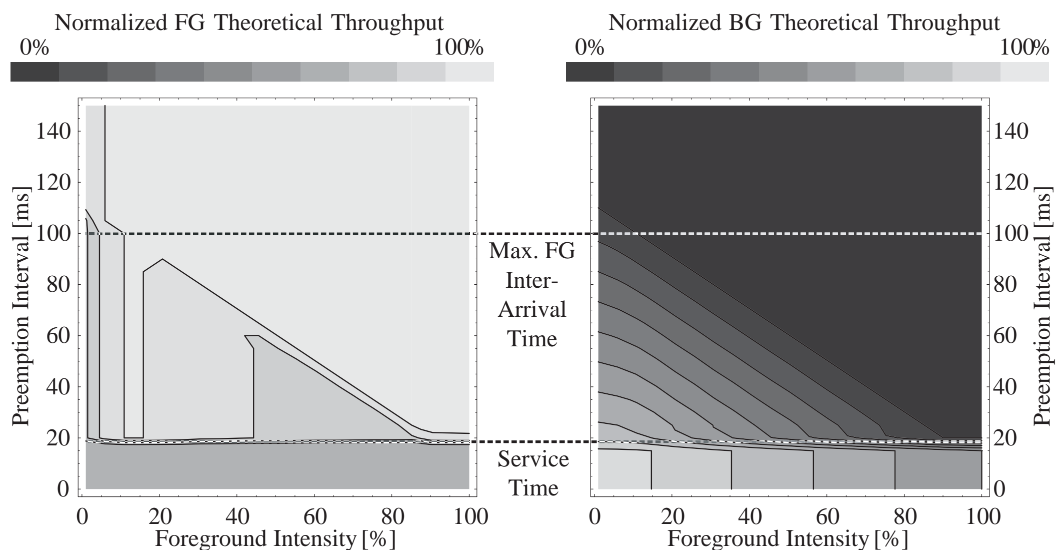


Figure 4.19. Predicted foreground (left) and background (right) performance of an idletime disk scheduler.

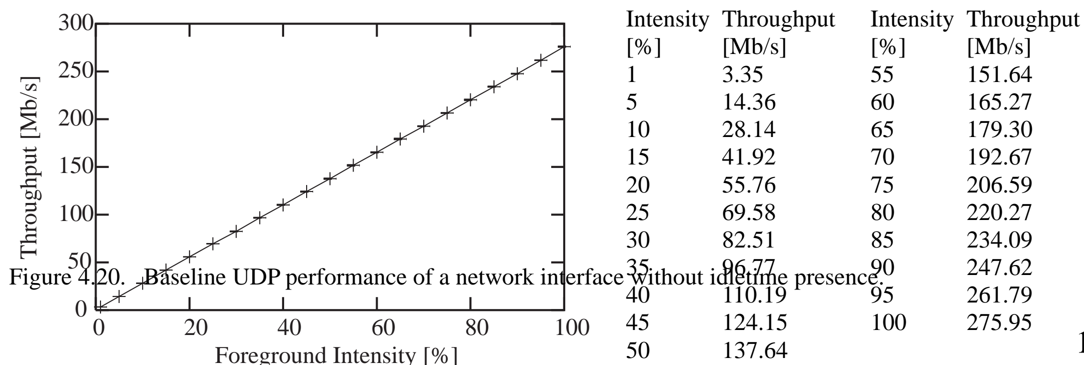
for low intensities in right graph). Note that the vertical bars on the bottom left indicate a performance “hump” between 5-40% intensity, with a peak at 15%.

Figure 4.19 illustrates two other interesting points. First, when the preemption interval is less than the service time and foreground intensities are high, the idletime scheduler degenerates into a simple priority queue. In this case, foreground and background both receive approximately 50% of the capacity. This indicates that priority queuing alone can be insufficient to establish idletime processing, even when high loads favor the formation of queues.

Secondly, Figure 4.19 illustrates how background processing is completely starved when the preemption interval exceeds the foreground inter-arrival time t_{max} of 100ms. The extreme case occurs at 90% intensity and higher, where this happens with a preemption interval that is only a few milliseconds longer than the service time.

4.4.3 Network Interface

This scenario predicts network send performance based on the setup shown in Figure 4.14. Here, the two load-generating processes send streams of UDP packets through



Intensity [%]	Throughput [Mb/s]	Intensity [%]	Throughput [Mb/s]
1	3.35	55	151.64
5	14.36	60	165.27
10	28.14	65	179.30
15	41.92	70	192.67
20	55.76	75	206.59
25	69.58	80	220.27
30	82.51	85	234.09
35	96.77	90	247.62
40	110.19	95	261.79
45	124.15	100	275.95
50	137.64		

Figure 4.20. Baseline UDP performance of a network interface without idletime presence.

the same network interface. The performance estimates presented in this section correspond to the experimental setup evaluated in Section 6.2.1.1 and will be compared against them in Section 7.1.1.1.

The baseline UDP send performance $r : I \rightarrow \mathbb{R}$ that describes performance in the absence of idletime use is based on measurements of the Intel PRO/1000F Fiber Gigabit Ethernet Adapter as displayed in Figure 4.20. This network card has a measured mean service time of 0.05ms at a packet size of 1500 bytes, including kernel processing.

Looking at the foreground performance (Figure 4.21, left graph) in detail, there are two regions of different performance. In the L-shaped area, performance is 60-70% of the baseline. In the rectangular area above it, performance is over 90% of the baseline.

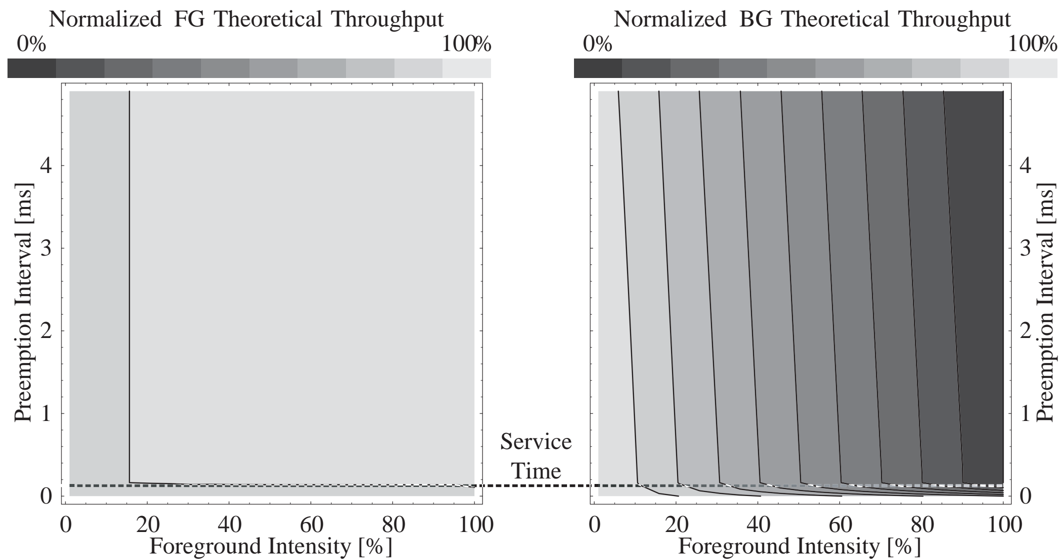


Figure 4.21. Predicted foreground (left) and background (right) performance of an idletime network scheduler with UDP traffic.

Inside the L-shaped area, the idletime scheduler is not effective. In the narrow horizontal part of the L, the preemption interval is shorter than the service time of 0.05ms. However, expected performance is also lower in the vertical part of the L, with intensities less than 17ms.

The right graph of Figure 4.21 shows the corresponding background performance. The gray stripes indicate that background performance gets progressively worse as foreground intensity increases, and effectively starves when foreground intensity reaches 100%. This is the desired behavior, and indicates that background performance is relatively high when foreground load is small – the idletime scheduler is effective in filling the available idle capacity with background work.

The background performance graph on the right of Figure 4.21 looks very different

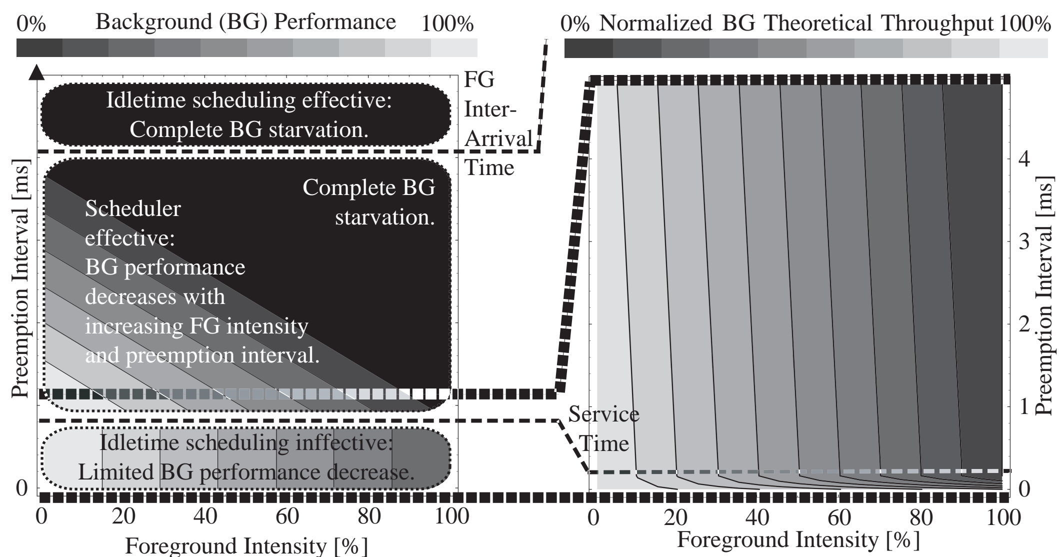


Figure 4.22. Visualization of the preemption interval range shown in Figure 4.21 (right) compared to the overview from Figure 4.17 (left).

from the overview shown in Figure 4.17. The latter showed two distinct triangular regions that split the middle area along its diagonal. In the top right triangle, background performance was extremely low, because idletime use stalled. In Figure 4.21, however this does not seem to occur.

The reason for this apparent discrepancy is that the service time of the network interface (0.05ms) is orders of magnitude less than the assumed inter-arrival rate of the foreground requests (100ms). The maximum preemption interval shown in Figure 4.17 is 5ms. Thus, Figure 4.17 shows only the small strip of the overview graph that lies right above the service time, where the effects of the preemption interval length on background throughput are not yet significant. Figure 4.22 visualizes this effect.

4.5 Summary

This chapter presented the detailed operation of an idletime scheduler based on a preemption interval, which bounds the preemption cost by relaxing the work conservation property. It formally defined the required properties in terms of a model that describes general resource processing, and presented a quantitative analysis that predicts the macroscopic behavior of the scheduler for different resources and workloads.

The next chapter discusses a prototype implementation in the FreeBSD operating system. Chapter 6 evaluates the performance of the prototype in a series of

experiments, and compares the measured performance against the results of the quantitative analysis presented in this chapter in Section 4.4.

5. Implementation

The previous chapter has given a detailed, formal description of the proposed idletime scheduler. This chapter will derive all algorithm variants that satisfy the conditions specified in Chapter 4 and discuss their specific characteristics. It will then identify the most promising variant as a basis for the implementation (presented in later sections of this chapter) and experimental evaluation (Chapter 6).

5.1 Scheduler Variants

This section will discuss the behavior of the idletime scheduler in terms of deterministic finite state machines. A resource with support for idletime scheduling using a preemption interval can be in four possible states. It can be idle (state *I*), process an active foreground request (state *F*), process an active background request (state *B*), or stall background processing during the preemption interval (state *P*). As a result, the set of states is $S = \{I, F, B, P\}$. The idle state *I* is always the start state.

The resource transitions between these states based on the four following events. First, a foreground request is at the head of the queue (event *f*). Second, a background

$$
\begin{array}{c} \\ I \\ F \\ B \\ P \end{array}
\begin{array}{c} \begin{array}{cccc} I & F & B & P \end{array} \\
\begin{pmatrix}
I \to I & I \to F & I \to B & I \to P \\
F \to I & F \to F & F \to B & F \to P \\
B \to I & B \to F & B \to B & B \to P \\
P \to I & P \to F & P \to B & P \to P
\end{pmatrix}
\end{array}
$$

Figure 5.1. Idletime state machine; adjacency matrix template.

request is at the head of the queue (event *b*). Note that due to priority queuing, event *b* cannot occur while a foreground request exists in the queue. Third, the preemption interval expired (event *t*). This can only occur when there is no *f*. Finally, the queue is empty (event *i*). Hence, the set of events is $E = \{f, t, b, i\}$.

The usual method of describing deterministic state machines is through an adjacency matrix, where each row describes the state transitions from a given state, and each column describes the state transitions into each state. For the given set *S* of states, Figure 5.1 shows the template for the corresponding adjacency matrix.

Each $X \rightarrow Y$ transition can occur on a given event $e \in E$. Clearly, a large number of state machines exist for the given set of events (4^{16}). The remainder of this section will reduce the number of variants to arrive at a small set of viable candidate mechanisms that conform to the idletime scheduling principles.

The first, obvious observation is that the resource shall remain in state *I* as long as it remains idle (event *i*). Secondly, Section 4.2.3 has identified strict work conservation for foreground requests as a requirement for idletime scheduling based on preemption

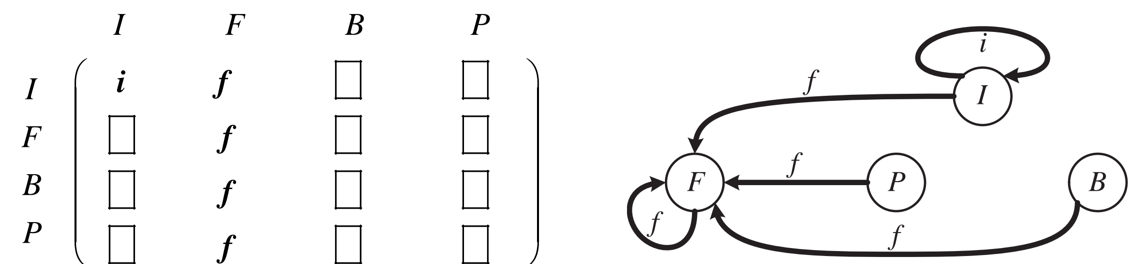


Figure 5.2. Idletime state machine with initial constraints.

intervals. It requires that the resource must immediately transition to state F when the head of the queue contains a foreground request (event f). These two observations constrain the adjacency matrix and result in this state diagram shown in Figure 5.2 (slots with rectangular placeholders are yet unspecified).

Another constraint is that a timeout event t will only happen during the preemption interval (when in state P). Repeated timeouts are not useful, ruling out the transition $P \to P$ on t. Because event t implies that there is no foreground request queued (*not* f), always transitioning to the idle state I on event t is the only useful choice. Furthermore, a preemption interval only starts after useful work, i.e., following state F or state B. They never follow state I, ruling out the $I \to P$ transition. This results in the state machine shown in Figure 5.3.

The key criterion for the idletime scheduler is that a preemption interval occurs when switching from foreground to idletime work. This means that all paths from state F to state B must go through state P (the preemption interval), eliminating $F \to B$ and $F \to I$. Consequently, state P must follow state F for both events b and i. Note that on event b, $F \to P$ does not consume b. The b/b notation signifies this in Figure 5.4.

$$
\begin{array}{c} \\ I \\ F \\ B \\ P \end{array}
\begin{array}{c}
\begin{array}{cccc} I & F & B & P \end{array} \\
\begin{pmatrix}
i & f & \square & - \\
- & f & \square & \square \\
\square & f & \square & \square \\
\mathbf{t} & f & - & -
\end{pmatrix}
\end{array}
$$

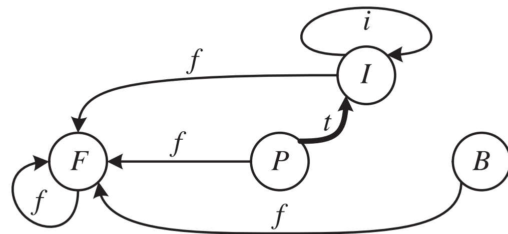


Figure 5.3. Idletime state machine with additional constraints.

Furthermore, entering state B requires an event b. Thus $I \rightarrow B$ can only occur on event b.

The resource can only process background requests that exist at the head of the queue; meaning that entering state B requires event b. This rules out $B \rightarrow B$ on event i. Also, $B \rightarrow I$ on b/b has no use, because $I \rightarrow B$ on event b would immediately follow. For event b in state B, the only two possibilities are therefore $B \rightarrow B$ or $B \rightarrow P$. Similarly, the only possibilities for event i in state B are $B \rightarrow I$ or (again) $B \rightarrow P$. Figure 5.5 shows all four possible variants, with the different transitions highlighted using thicker arcs.

The four state machine variants in Figure 5.5 all satisfy the idletime properties defined in earlier sections of this chapter. They are prioritized, preempting, and strictly work conserving for foreground requests, and weakly work conserving for idletime requests. Whenever the resource switches from foreground to idletime use, it incurs a preemption interval. In terms of the state machine, this means each path from F to B will visit P.

	I	F	B	P
I	i	f	$\boldsymbol{b}$	–
F	–	f	**-**	$\boldsymbol{b/b \vee i}$
B	□	f	□	□
P	t	f	–	–

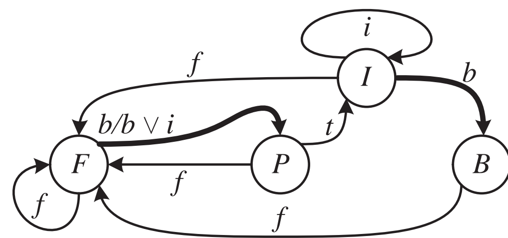
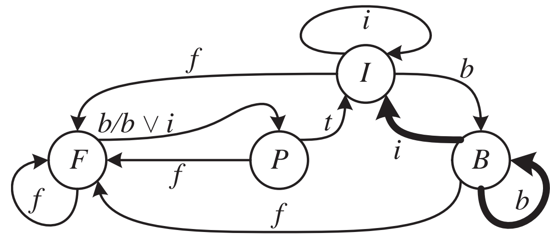


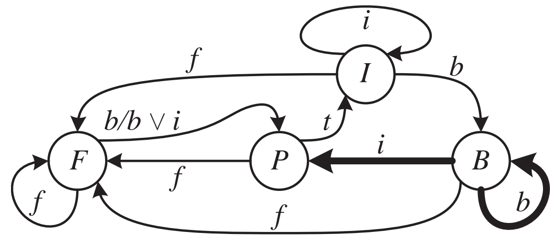
Figure 5.4. State machine after third set of constraints.

However, several differences exist between the four variants. The first two variants in Figure 5.5 will remain in state *B* for a burst of *b* events, whereas the second two will switch to state *P* and require a timeout on each *b* event. The difference is that the first two variants will incur a single preemption interval before each burst of idletime requests: $b^{+} \rightarrow IB^{+}$. The second two will instead incur a preemption interval before

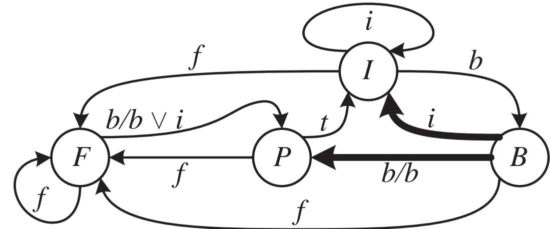
$$\begin{array}{c} \\ I \\ F \\ B \\ P \end{array} \begin{array}{c} \begin{array}{cccc} I & F & B & P \end{array} \\ \left(\begin{array}{cccc} i & f & b & - \\ - & f & - & b/b \vee i \\ \mathbf{i} & f & \mathbf{b} & \mathbf{-} \\ t & f & - & - \end{array} \right) \end{array}$$



$$\begin{array}{c} \\ I \\ F \\ B \\ P \end{array} \begin{array}{c} \begin{array}{cccc} I & F & B & P \end{array} \\ \left(\begin{array}{cccc} i & f & b & - \\ - & f & - & b/b \vee i \\ \mathbf{-} & f & \mathbf{b} & \mathbf{i} \\ t & f & - & - \end{array} \right) \end{array}$$



$$\begin{array}{c} \\ I \\ F \\ B \\ P \end{array} \begin{array}{c} \begin{array}{cccc} I & F & B & P \end{array} \\ \left(\begin{array}{cccc} i & f & b & - \\ - & f & - & b/b \vee i \\ \mathbf{i} & f & \mathbf{-} & \mathbf{b/b} \\ t & f & - & - \end{array} \right) \end{array}$$



$$\begin{array}{c} \\ I \\ F \\ B \\ P \end{array} \begin{array}{c} \begin{array}{cccc} I & F & B & P \end{array} \\ \left(\begin{array}{cccc} i & f & b & - \\ - & f & - & b/b \vee i \\ \mathbf{-} & f & \mathbf{-} & \mathbf{b/b \vee i} \\ t & f & - & - \end{array} \right) \end{array}$$

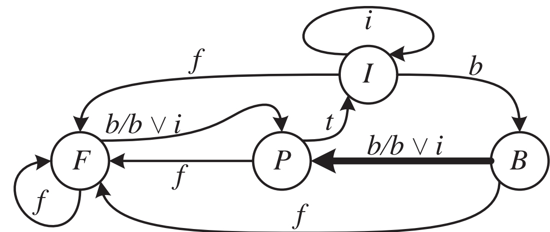


Figure 5.5. Four possible variants of the idletime state machine.

each idletime request in a burst: $(bt)^{+} \rightarrow I(BPI)^{+}$. In some sense, the first two variants are strongly work conserving for idletime requests once state *B* is reached and idletime use begins, whereas the last two variants are always weakly work conserving for idletime requests.

Idletime performance of the two variants that are strongly work conserving for the idletime workload will be significantly higher than for the weakly work conserving variants. On the other hand, the additional preemption intervals before each idletime request enforced by the weakly work conserving variants increases the likelihood that arriving foreground requests will find the resource idle. They can consequently further decrease foreground delays. Because the purpose of a preemption interval is delaying idletime use after foreground use, enforcing additional preemption intervals between idletime requests – when the last foreground request could have happened long ago – is not likely to be useful.

The second difference between the variants is their behavior when idletime use is bursty, i.e., when *i* events occur between *b* events. The first and third variant in Figure 5.5 will immediately enter the idle state *I* when event *i* is encountered during idletime

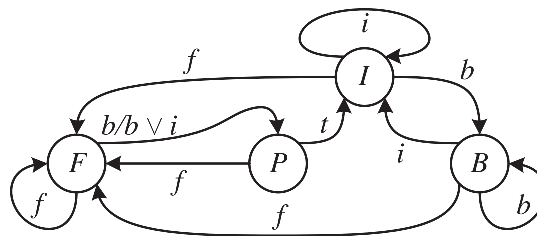


Figure 5.6. Variant of the idletime scheduler chosen for implementation.

use in state B: $(bi)^{+} \to I(BI)^{+}$. The second and fourth variants enter a preemption interval P instead, and require additional timeouts: $(bit)^{+} \to I(BPI)^{+}$. Extra preemption intervals between two idletime bursts will therefore decrease idletime performance, and in turn reduce foreground delays. As before, however, enforcing additional preemption intervals between bursts of idletime use – without occurring foreground use – is not likely to be useful.

Therefore, the first variant – shown in Figure 5.6 – was chosen for implementation and experimental evaluation (see Chapters 5 and 6). It maximizes idletime performance by avoiding preemption intervals between successive idletime requests, as well as between bursts of idletime requests. Because maximizing idletime use was the secondary objective of a successful idletime mechanism (prevention of foreground delays is the primary objective), the first variant appeared most functional.

5.2 Implementation Overview

This section will discuss the prototype implementation of idletime scheduling for two different resources, a disk drive, and a network interface. Both are simple, localized modifications to release 4.7 of the FreeBSD operating system. Chapter 6 will present an extensive experimental evaluation of the prototype, and compare it to the predicted performance derived from the model in Section 4.4.

The prototype implementation tags each resource request as either regular or idletime. Idletime schedulers in the system use the tags to prioritize the request stream according to idletime properties. The prototype implementation defines a new idletime option for file descriptors (including sockets), that indicates whether resource operations occur in the foreground – the default – or background. A process can explicitly set and clear the idletime option to start or stop issuing background requests, respectively. While the option is set, the kernel tags all operations on the descriptor as idletime.

A new file descriptor option requires application changes to use idletime capacity. However, other alternative idletime APIs, to allow unmodified applications to execute during idletime, are possible. One example would be to simply overload the meaning of CPU priority, and treat all resource requests from processes running with less than a specific CPU priority as idletime. Another variant is redefining the POSIX idletime

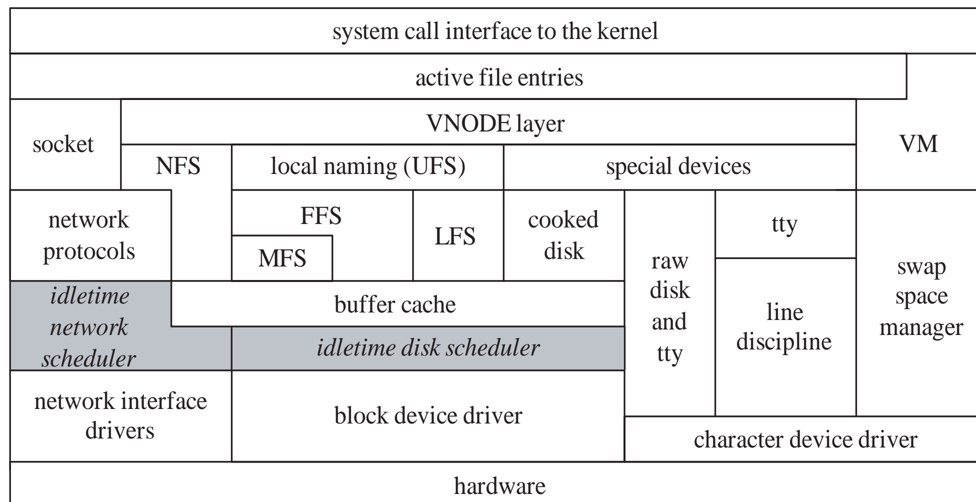


Figure 5.7. FreeBSD kernel I/O overview. Adapted from [McKusick1996].

flag – originally defined for CPU scheduling – as a general idletime flag covering other resources [POSIX1993]. Finally, for idletime networking, specific, reserved port ranges or IP aliases could indicate idletime use.

Both the modified network and disk schedulers implement the preemption interval mechanism with the standard BSD timing facilities [VARGHESE1997]. A separate timer is associated with each network and disk device. The timer restarts whenever the scheduler for the given resource reenters state *P* (see Section 4.3). While the timer is active, the resource is in its preemption interval (resource condition *q* is satisfied). Upon timer expiration, the resource either starts serving background requests (entering state *B*) or becomes idle (state *I*).

Neither prototype implementation modifies the device drivers of the resource for which they implement idletime use. This is important, because idletime use would otherwise incur a significant deployment issue due to the multitude of drivers in a typical system that would all require modifications. Instead, the prototype implementations are simple, localized modifications at a higher system layer. In Figure 5.7, the idletime disk scheduler operates at the border between the buffer cache and the block device driver, and the idletime network scheduler operates at the border between the network protocols and network interface drivers.

5.3 Idletime Disk Service

To issue background disk requests, a process uses the *fcntl* system call to set the idletime option on an open file descriptor. The kernel will then tag all underlying block transfers for idletime scheduling at the buffer queue.

Figure 5.8 shows an overview of the processing involved in disk I/O, using the *read* system call as an example. (The *write* operation is similar but simpler, and does not exhibit all of the issues of the *read* case).

The native file system in FreeBSD is UFS, a variant of the original Berkeley Fast File System [MCKUSICK1984]. For UFS, the corresponding *vnode* operations check whether the requested block resides in the cache. When it does, the kernel copies its contents to the application buffers and process execution resumes without any physical disk I/O. If the block is not cached, the kernel schedules physical disk I/O (clustered or unclustered), signals the device, and switches to other tasks until I/O completes.

Meanwhile, the device driver dequeues a waiting I/O request from its work queue, issues the request to the underlying hardware, and signals completion to the waiting *vnode* operation after the disk request finishes. After copying the data into the application-provided buffer, as well as adding the retrieved block to the buffer cache, process execution resumes.

The current system implements idletime scheduling by replacing the standard *bufqdisksort* algorithm, which normally implements the *C-LOOK* variant of the *elevator seek* algorithm [WORTHINGTON1994]. The prototype replaces this with two queues for foreground and background requests, each managed by the original *C-LOOK* variant to establish prioritization, plus the preemption interval mechanism described in Chapter 4.

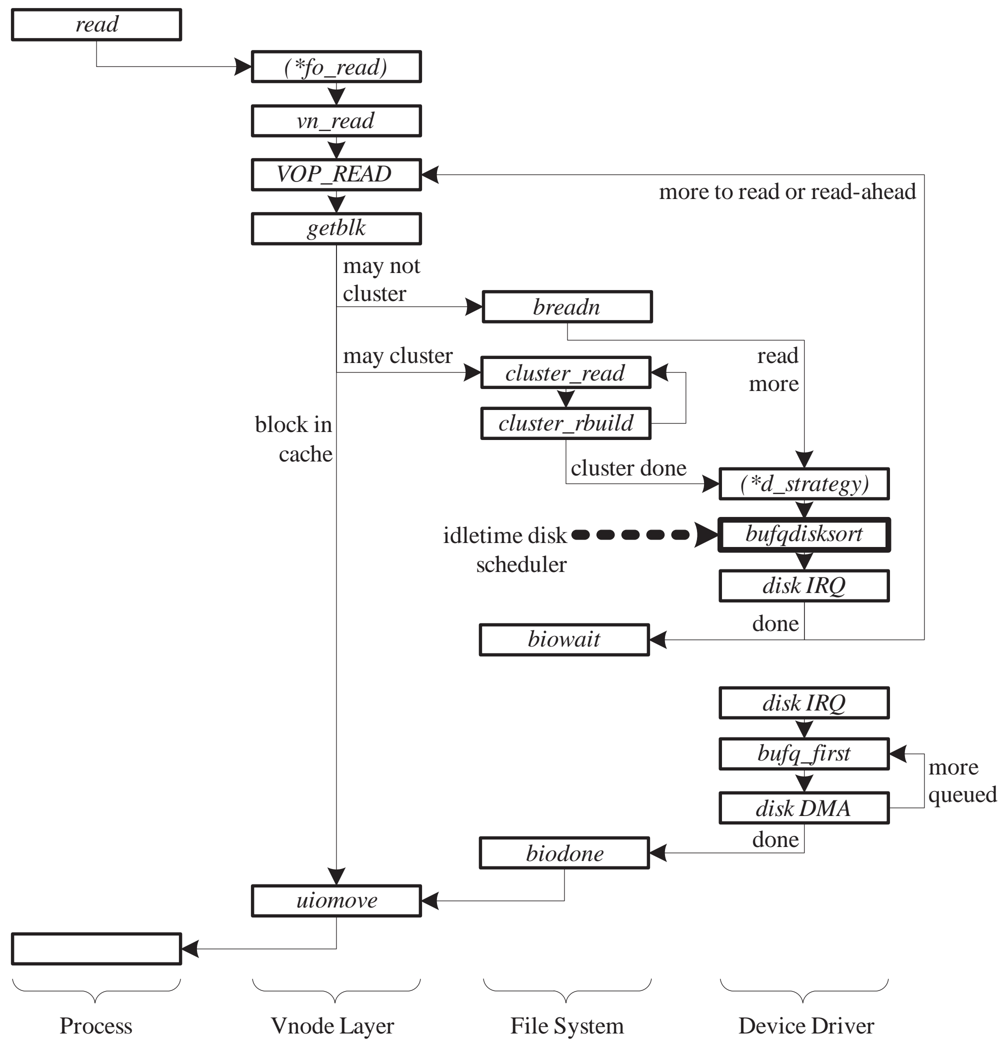


Figure 5.8. FreeBSD disk I/O processing.

Many modern disk drives service multiple concurrent operations, and manage the processing order internally in hardware. Often, the on-disk hardware queue does not support request priorities, and many drives reorder pending foreground and idletime operations to lower the combined service time. This is another example illustrating why a simple priority-based scheme does not support effective idletime use when some resources remain unmodified (see Section 4.3.2). Preemption intervals can counteract these effects, and still support idletime use. The device driver will stall idletime requests during the preemption interval and prevent them from entering the hardware queue. This way, they cannot interfere with regular operations on-disk, decreasing the impact of idletime use on foreground performance.

One performance-enhancing feature of the UFS file system is its buffer cache, as previously described. A second feature is read-ahead: during *breadn* or *cluster_read*, the file system uses a heuristic to predict whether the most recent request history corresponds to a sequential read operation. When it does, the file system will speculatively generate additional read operations for the next disk blocks. When predicted correctly, these blocks will already reside in the buffer cache when the application requests them, improving performance. Furthermore, read-ahead of larger chunks can sometimes take advantage of more efficient bulk I/O commands, further increasing performance.

For idletime use, both caching and speculative read-ahead are problematic. First, the buffer cache is of limited size. Using it to hold idletime data can decrease regular

foreground performance, if it flushes foreground data from the cache. Section 2.3.5 already discussed cache pollution in detail, and explained how treating the cache itself as a spatially shared resource would support idletime use. The prototype implementation does not support such idletime caching. Instead, the prototype completely disables caching of idletime data – the buffer cache only holds foreground data. Only the buffer cache inside the operating system was disabled. On-disk hardware caches remained active and could lead to reduced performance when the disk drive flushed buffered foreground data to cache new idletime information.

Read-ahead is also problematic for idletime service, because it can further decrease foreground performance by causing additional preemption costs. When the file system determines that an idletime process is performing a sequential read, it schedules many additional speculative read-ahead operations. These speculative I/O operations will receive service along with the single application-requested I/O operation, and lengthen the duration of idletime use of the device. This will delay new foreground requests arriving during this time, and lower foreground performance. The prototype implementation accordingly disables read-ahead completely for idletime use.

Some aspects of the idletime disk scheduler are similar to other research. McKusick's proposes decreasing the aggressiveness of a background *fsck* operation that repairs file systems after a crash [McKusick2002]. Background *fsck* delays each disk request of processes with positive nice values (low priority) whenever the disk queue is not empty. The specific delay time depends on the *nice* level of the issuing process. This

scheme is similar to the idletime variant discussed in Section 5.1 that enters a preemption interval after each background request, and suffers from the same low throughput.

Jeff Roberson's *prio* patch [ROBERSON2002] changes FreeBSD's disk scheduler from using a FIFO to a two-level priority queue. It does not include preemption intervals, and therefore suffers from decreased foreground performance under high idletime loads due to frequent preemption costs.

A third proposal is the anticipatory disk scheduler [IYER2001], which determines delay times for each disk request to improve spatial and temporal locality among requests from different processes. It proposes a concept similar to preemption intervals that keep the disk idle for short periods of time while requests are queued, relaxing work conservation. The anticipatory scheduler focuses on improving locality of reference – and hence performance – through these stall times, by giving process the opportunity to generate additional requests for disk blocks near the current head position. Unlike the proposed idletime scheduler, the anticipatory disk scheduler is specifically tailored to disk resources (hence the name), does not support priority levels, and can as a result not support idletime use.

Disk drives are stateful resources, where a previous request can affect the processing time of the current one because of disk head location. This violates the isolation property, as disk head location is a side affect that could delay foreground

performance. Some proposals discuss disk head relocation during idletime [KING1990][MUMOLO1999]; no such mechanism was implemented for the initial idletime scheduling prototype. However, disk head relocation and similar mechanisms to mask the presence of idletime use are one direction for future research.

5.4 Idletime Network Service

Network support for idletime use is more complex than the disk prototype discussed in the previous section. Unlike disk accesses, idletime networking requires support at remote systems along the paths idletime packets take through the network. The first part of this section will discuss these requirements in detail, whereas the second part focuses on the prototype implementation.

5.4.1 Networking Overview

Idletime networking is a simple extension of the current Internet service model, where routers (and hosts) treat packets equally according to a best-effort discipline [CLARK1988]. Its fundamental principles remain, and the Internet may still reorder, drop, or duplicate packets. Idletime service is strictly a per-hop function of giving higher processing or bandwidth preference to certain packets. This is not a new idea: the original IP specification [POSTEL1981] contains support for a precedence field in the datagram header to indicate forwarding priorities.

More recently, some of the proposed extensions to support differentiated services in the Internet [BLAKE1998] are similar to idletime networking. Expedited forwarding [DAVIE2002] redefines a value in the IP type-of-service field to mark some packets with a higher forwarding priority. It also suggests configuring a rate limit for expedited packets, in order to prevent starvation of lower-priority traffic. Although its focus is on providing provisioned virtual links, it can also support idletime networking by sending all regular traffic using expedited packets, and idletime data with regular best-effort packets.

A combination of two other proposals from the differentiated services community is also similar to idletime networking. One proposal has routers mark packets as *in* or *out* of compliance with their assigned traffic class [CLARK1998]. During congestion, packets marked as *out* are give drop preference (similar to ATM's cell-loss-priority bit [ATM1999] or frame relay's discard-eligible bit [THIBODEAU1998]). The second proposal has routers forward packets in strict order of priority [GUPTA1997]. Together, these proposals support idletime use by giving drop preference and lower forwarding priority to idletime packets. [CARLBERG2001] and [MAY1997] propose further service models based on these mechanisms. Section 8.4 discusses this and other related research in more detail.

All of these proposals focus on router support for their various service schemes. Prioritization is an important component of idletime use. For busy routers with long queues, priority queuing alone may be the critical property supporting idletime use.

However, Section 4.3 showed how prioritization alone is ineffective for light workloads. This applies to lightly loaded routers as well, where preemption costs can delay foreground use, even with priority queuing. A queue scheduler with a preemption interval can further minimize these delays.

One approach to decrease preemption costs for network transmission uses small packet sizes [BORMANN1999]. Because packet transmissions are usually not preemptable, smaller packet sizes allow switching service to newly arriving, higher-priority traffic more frequently. Although originally targeted at low-bitrate links where preemption costs are high for long packet sizes, it may also improve foreground performance when applied to idletime traffic on higher-speed links. However, small packet sizes significantly increase the interrupt load on end systems, which can decrease overall system efficiency [MOGUL1997]. Furthermore, most routers are packet-rate-bound, and may sustain lower overall throughput with smaller packet sizes.

Idletime scheduling with a preemption interval is also important for end systems. Routers perform very limited operations at layers 2 and 3, and can therefore support idletime service more easily than end systems, with very complex operations at all layers of the protocol stack. The next section will discuss the current prototype for idletime networking in detail.

5.4.2 Prototype Implementation

To start sending idletime traffic, a process uses the *setsockopt* system call on an open socket to set the idletime option. The idletime scheduler operates at the network layer. The kernel tags IP packets sent from a socket that has the idletime option set with a particular type-of-service value. The current implementation uses *0x20*, which the Internet2's Q-Bone Scavenger Service proposes for a similar purpose [SHALUNOV2001]. The current implementation supports IPv4, but IPv6 support is straightforward using the traffic class field in the IPv6 header.

When receiving a packet with the type-of-service field set to idletime, the kernel will set the idletime option on the corresponding receive socket. This causes future application responses on the same socket to use idletime traffic automatically also. A future release will make this behavior optional, to give applications explicit control over their service level.

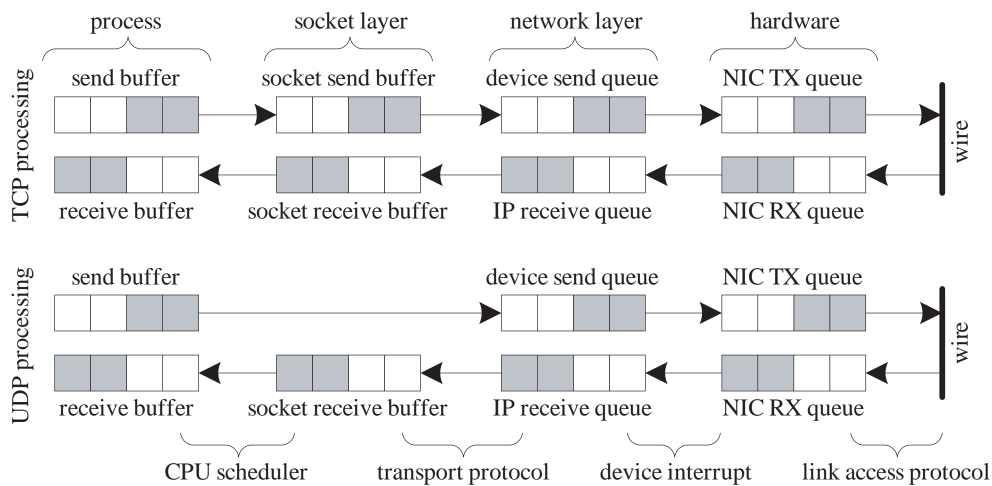


Figure 5.9. Queuing at different layers in the network stack.

The preemption interval scheduler for network idletime service extends the *ALTQ* queuing framework [CHO1998], which is a part of the *KAME* IPv6/IPsec implementation [JINMEI1998]. *ALTQ* replaces the standard FIFO outbound queue (device send queue in Figure 5.9) with configurable queuing disciplines, including a priority queue, which served as the basis for the idletime scheduler extensions.

ALTQ only modifies outbound queues. However, systems also queue inbound packets during receive processing, using a FIFO by default (see Figure 5.9). An earlier research effort produced an *ALTQ* extension to support different queuing disciplines for the inbound queue [EGGERT2001A]. However, the idletime inbound module is a modification of an earlier version of *ALTQ*, and was hence not used for this prototype implementation.

Figure 5.10 shows a simplified overview of processing in the network stack of the FreeBSD operating system. The idletime scheduler replaces the indirect *ifq_dequeue* call all network drivers make to obtain the next packet to send. In the original implementation using a FIFO queue, this call would simply return the packet at the head of the queue.

Unlike disk scheduling, no known implementation of a network scheduling mechanism exists that relaxes the work conservation principle. Most research into providing different priority levels of network service focuses on routers [CLARK1998][SHALUNOV2001].

Routers forward traffic but do not source or sink significant amounts of traffic themselves. Their processing environment is significantly simpler compared to an end system. Furthermore, routers are highly optimized for packet forwarding at line rates, and their bottleneck resources are engineered to be the egress interfaces. As a result, effective priority mechanisms for routers can focus on queue management alone, and need not consider effects of higher-level processing required at an end system.

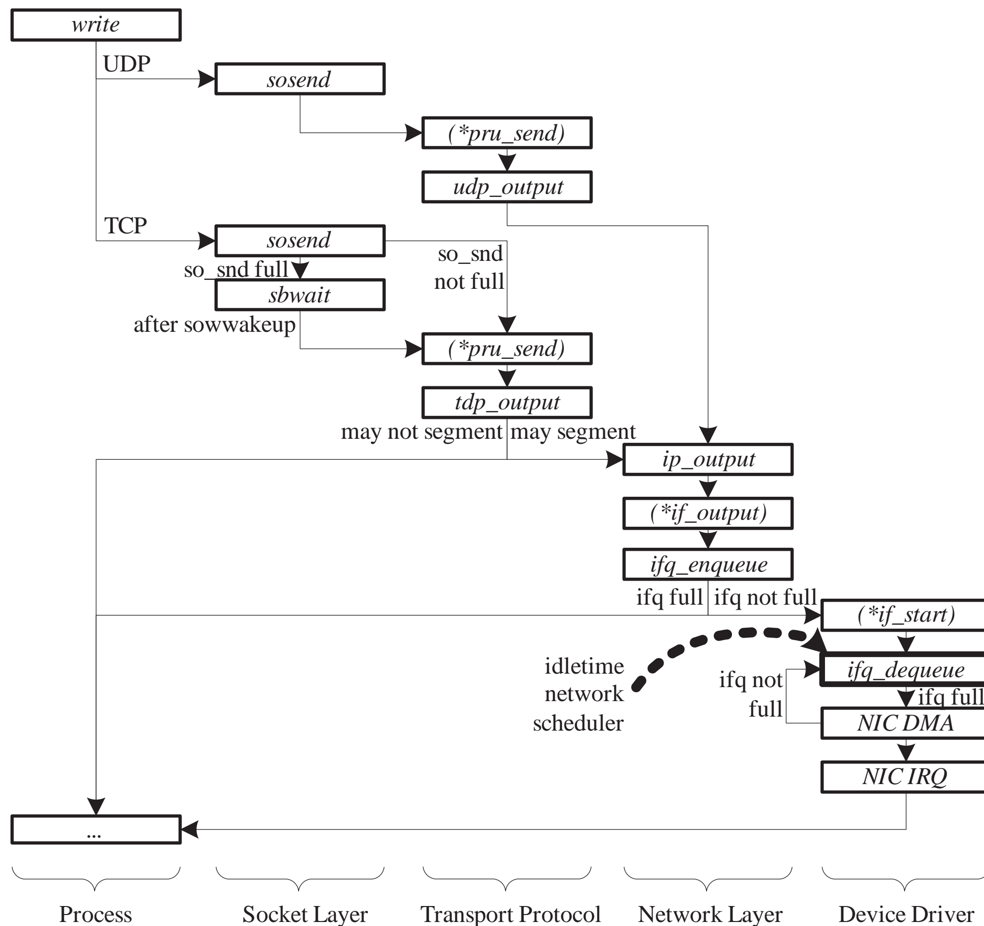


Figure 5.10. FreeBSD network stack processing.

5.5 Implementation Considerations

One key difference between the prototype implementation and the initial model presented for idletime service in Chapter 4 is the starting point of the preemption interval. The prototype implementation starts the preemption interval timer when the resource starts processing a foreground request, instead of immediately after it finishes.

This difference to the model was a design decision, and allows investigation of scenarios where the preemption interval length is shorter than the service time of the resource. This also simplifies the implementation for resources that internally batch together or reorder requests in hardware. They would otherwise require individual driver modifications instead of a single modification at a higher layer. The quantitative analysis in Section 4.4 already assumed that a preemption interval starts at the beginning of a foreground request, to allow for easy comparison of the predicted and measured performances (see Section 7.1.1).

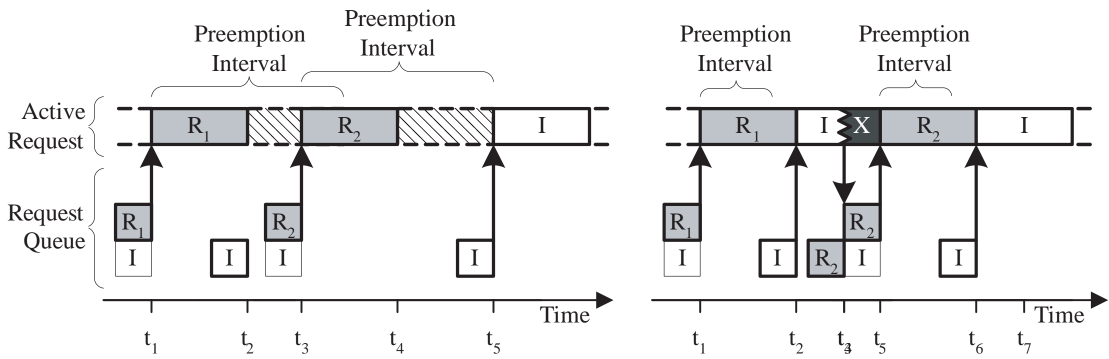


Figure 5.11. Effects of preemption interval length when preemption intervals start at the beginning of foreground requests.

Figure 5.11 illustrates this difference to the model (left diagram), and shows how preemption intervals shorter than the service time of the resource become ineffective (right diagram). This happens, because the preemption interval expires while its corresponding foreground request is still active, and the idletime scheduler degenerates into a priority queue.

The preemption interval timers in the prototype implementation use the standard FreeBSD timing facilities [VARGHESE1997], which offer very efficient, constant-time timer operations. However, the current prototype scheduler incurs a timer restart for each foreground request. The timer management overhead could become noticeable for very high-rate resources. A first improvement is to batch foreground requests together and only restart the timer at the end of a batch. Additionally, improvements to the timer facility itself can further decrease this overhead [ARON2000]. Very high-rate resources may require direct use of hardware timers.

One limitation of the current prototype is that the length of preemption interval is a per-scheduler variable for each scheduler. This means that all network or disk idletime use at a given time is based on the same preemption interval length. Although this does not affect the validity of the benchmarks presented in Chapter 6 – because they only measure idletime use on a single resource – a future revision must address this issue. The length of the preemption delays is clearly a per-resource property, and will need to be maintained as such by the system.

Other proposed system optimizations could also further improve the idletime scheduler. One example is *lazy receiver processing (LRP)* [DRUSCHEL1996]. *LRP* demultiplexes the incoming packet stream at the link layer into channels according to their destination socket. It then performs receiver protocol processing at application priority. Its main goal is increasing system fairness and stability under high traffic loads by shortening the time spent in response to device interrupts. Without a mechanism like *LRP*, the system can enter livelock under high network loads [MOGUL1997], where it spends all of its cycles during interrupt processing, and higher-layer processing effectively stalls.

5.6 Summary

This chapter described the possible variants for an idletime scheduler based on the properties identified in Chapter 4, and identified the best possible variant for prototype implementation. To establish idletime disk service, an idletime scheduler using preemption intervals replaced the traditional *bufqdisksort* algorithm. Idletime networking operates as a similar extension to the *ALTQ* queuing framework. Applications use these new capabilities through an extended file descriptor interface that offers idletime service as a new per-descriptor option.

The next chapter will experimentally evaluate these prototype implementations, and compare their performance against the predicted behavior, based on the quantitative analysis in Section 4.4.

6. Evaluation

The previous chapter described the prototype implementation of the idletime scheduler defined in Chapter 4. This chapter provides a detailed experimental investigation of the measured performance of the prototype under various loads and discusses the results. Later sections of this chapter will then compare the experimental performance numbers with the predicted performance based on the quantitative analysis in Section 4.3.1.

All experiments in the remainder of this section follow the same basic setup shown in Figure 6.1, which is identical to the scenario modeled in the quantitative analysis in Section 4.3.1. Two benchmark processes run concurrently, one issuing foreground requests, the other issuing background requests for the same resource. The benchmark processes measure the perceived throughput and latency for each request.

The intensity of the foreground is one variable of the experiment and varies between 1-100%. Foreground intensity specifies the fraction of each CPU quantum a process spends generating resource requests. For example, at 50% intensity a process spends half its cycles generating load and half its cycles performing other tasks. Varying the

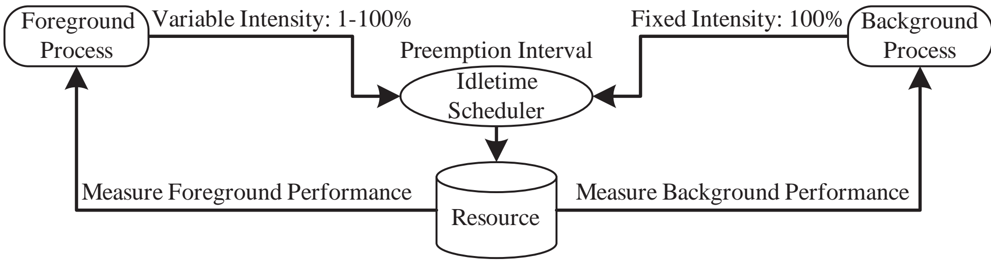


Figure 6.1. Experimental setup.

foreground intensity allows to study the impact of idletime processing as the foreground workload increases. For this purpose, the exact foreground request patterns generated through this approach are not relevant.

The background process is greedy, and tries to consume as much resource capacity as possible. The intention of this work is to utilize any idle resource capacity for useful work, having the background process spend all its cycles generating resource requests models this scenario. It is important to note this models the worst-case scenario for idletime use, because each time a new foreground request arrives, it incurs a potential preemption delay due to ongoing idletime use. The resulting performance measurements therefore also determine the worst-case performance of the scheduler.

A second variable of the experiment is the length of the preemption interval. Depending on the resource, it varies from zero up to a few hundreds of milliseconds. When the preemption interval is zero, the behavior of the scheduler is identical to a simple priority queue. A preemption interval larger than the CPU quantum (100ms) completely stalls idletime use. The benchmark processes will generate at least one resource request per CPU quantum by design, and the preemption interval can therefore never expire.

The preemption intervals used by both prototype schedulers start at the beginning of each foreground requests, not at its end. This difference to the model (see Chapter 4) is

due to implementation limitations described in Chapter 5, and similar to the quantitative analysis in Section 4.4.

Each experiment consists of five separate 30-second iterations for each data point (unique pair of intensity and preemption interval) to compute the mean foreground and background performances and their standard deviations. Standard deviations were usually well below 5% and were therefore omitted to avoid cluttering the contour plots in the remainder of this chapter.

Normalizing the mean measured performances against the performance of the baseline case (without background work present) allows comparison of the relative impact background processing. Again, this is the same normalization used during the quantitative analysis in Section 4.3.1, and the remainder of this section will present the same contour plots used before.

It is important to consider both throughput and latency numbers when evaluating the effectiveness of an idletime scheme. For each experiment, the following sections will present four contour plots side-by-side, showing the normalized mean foreground and background throughputs and latencies. Lighter shades of gray in the plots indicate areas of better performance (higher throughput or lower latency).

Figure 6.2 gives a rough overview of the expected contour plots for an ideal idletime scheduler, and identifies regions of interest. For example, the scheduler would support

close to 100% foreground performance with a preemption interval larger than the resource service time, independent of the foreground intensity. This appears as the larger, lightly shaded area in the left plot in Figure 6.2. The dashed line cutting across both plots signifies the service time of the resource. With a preemption interval of less than the service time, the scheduler is ineffective (bottom, darker area in the left plot). For reference, the baseline case without idletime processing would appear as a single, evenly shaded area of the lightest shade of gray, independent of foreground intensity or preemption interval lengths.

The right plot in Figure 6.2 shows the corresponding background performance in the same scenario. An ideal idletime scheduler will utilize available capacities for background work (especially at low foreground intensities), but will begin to starve background work as foreground intensities rise. Consequently, background performance will decrease with increasing foreground intensity and preemption

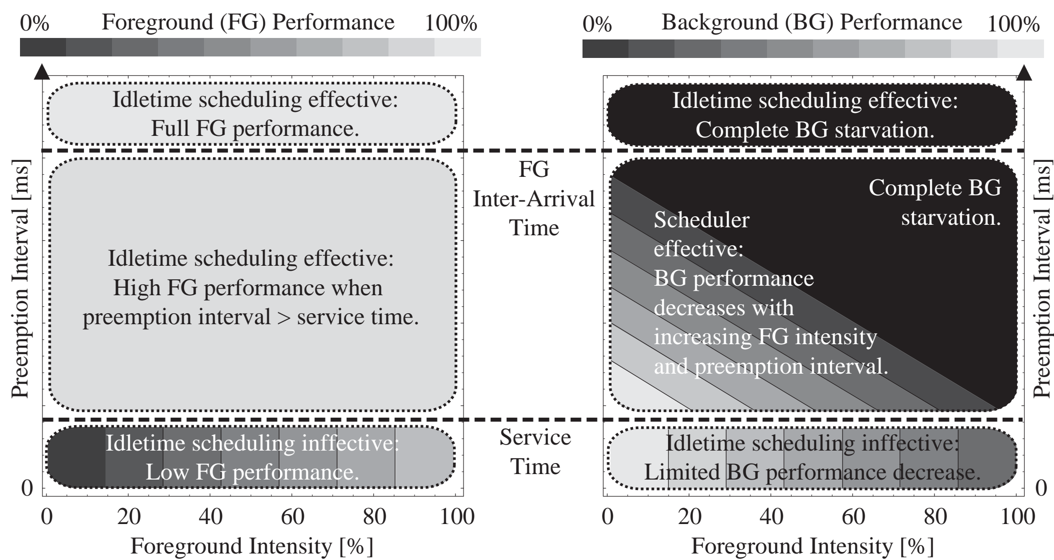


Figure 6.2. Overview of the expected performance behavior for an idletime scheduler.

interval lengths, as illustrated by the gradient in the larger area in the right plot. With a preemption length shorter than the service time, the mechanism is again ineffective. Background performance will still decrease with higher foreground intensities, but will not become completely stalled (smaller, bottom area in the right plot of Figure 4.17), and foreground performance will decrease.

Section 2.1 described the two main criteria for evaluating the effectiveness of an idletime scheduler: impact on foreground processing, and amount of background work scheduled. An ideal idletime scheme will exhibit performance identical to the baseline case across the full range of foreground intensities, for preemption intervals longer than a specific resource-dependent bound. In the contour graphs, this will show as very light shades of gray. It will also succeed in scheduling some background work during idletime, especially at light foreground intensities. This will show in the graphs as lighter areas.

The experiments used PC workstations running release 4.7 of the FreeBSD operating system together with a *KAME* [JINMEI1998] snapshot release that included *ALTQ* [CHO1998], modified to support background processing as described in Chapter 5. Each PC was equipped with 512MB of RDRAM and dual 733MHz Intel Pentium-III processors. FreeBSD 4.7 can run user processes simultaneously on multiple processors, but only allows a single CPU to execute kernel code at any time. This eliminates contention between the foreground and background load generators – increasing the offered load – without affecting in-kernel processing.

6.1 Disk Scheduler Evaluation

During the disk benchmarks, the foreground and background benchmark processes generate fixed-size disk read requests on the same test file containing random data. The 8.2GB test file spans a UFS file system that completely utilizes a Western Digital Caviar AC28200 disk connected on a separate ATA channel. The drive has a manufacturer-reported maximum mean seek time of 15ms and a mean latency of 5ms, including controller overhead, resulting in a 20ms total mean service time for a random access to a disk block.

Two separate experiments, presented in the remainder of this section, investigate the performance of the idletime scheduler under random and sequential disk accesses. In the first scenario, the benchmark processes read single bytes from random locations across the test file, causing accesses to single random disk blocks. In the second scenario, the processes read 512-byte disk blocks sequentially from the test file. The benchmark processes start reading at different offsets into the test file to eliminate buffer cache overlap. The benchmark processes also re-mount the test file system before each run, which flushes the buffer cache and eliminates cache effects across successive runs.

6.1.1 Random Access

In this experiment, both foreground and background process read 512-byte disk blocks from random locations in the 8.2GB test file. During each run, they read up to 1,600

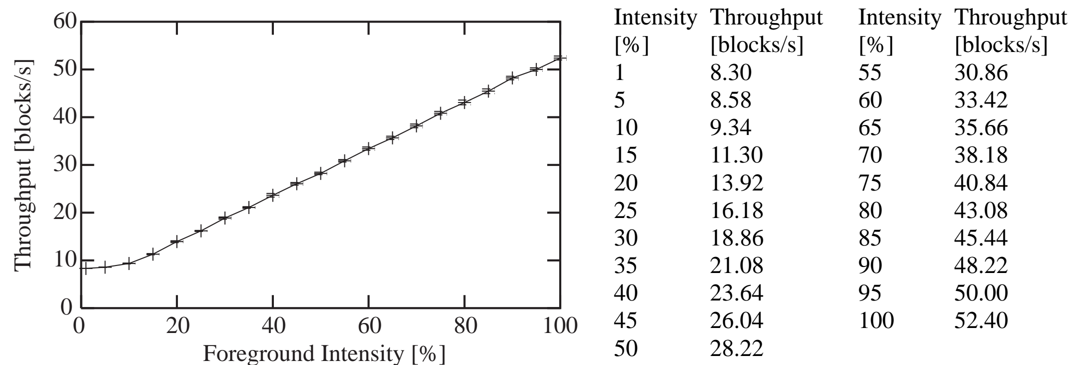
blocks at a rate of up to 60 blocks/second. Figure 6.3 shows the detailed baseline performance without idletime presence used for normalization. The error bars indicate standard deviation, which is below 0.5 blocks/s in all cases.

Figure 6.4 shows the throughput (top) and latency (bottom) of the foreground (left) and background (right) benchmarks. Lighter shades of gray indicate areas of better performance (higher throughput or lower latency). The graphs break down in to three major areas: less than 20ms (preemption interval less than service time), greater than 100ms (preemption interval greater than CPU quantum), and the middle range of preemption intervals between 20-100ms.

With a preemption interval of less than 20ms, the idletime scheduler has no effect. The background benchmark monopolizes the disk, receiving 50-100% throughput, while the foreground throughput degrades to 50% with increasing intensity. Latency of foreground reads is at least a factor of 1.5 higher than the baseline case, while background read latencies are low.

Figure 6.3. Baseline disk read performance without idletime presence under a random-access workload.

This ineffectiveness of the idletime scheduler is expected with preemption intervals



Intensity [%]	Throughput [blocks/s]	Intensity [%]	Throughput [blocks/s]
1	8.30	55	30.86
5	8.58	60	33.42
10	9.34	65	35.66
15	11.30	70	38.18
20	13.92	75	40.84
25	16.18	80	43.08
30	18.86	85	45.44
35	21.08	90	48.22
40	23.64	95	50.00
45	26.04	100	52.40
50	28.22		

shorter than the service time. The mean access delay (seek time plus latency) of the drive in this benchmark is approximately 20ms. A preemption interval shorter than its service time expires while its corresponding foreground request is still active. Thus, any enqueued background request will immediately receive service after processing of the last foreground request finishes. The behavior is hence identical to a traditional priority queue.

However, although preemption intervals shorter than the service time fail to address foreground performance degradation, they do affect corresponding idletime performance.

The top right graph in Figure 6.4 illustrates that idletime throughput already starts to decrease with preemption intervals longer than 10ms. Foreground throughput, however, does not benefit from this reduction of idletime load, until the preemption intervals exceed the service time.

The second region of interest has preemption intervals of more than the CPU quantum of 100ms. Here, the idletime scheduler almost completely suppresses processing of background requests. The preemption interval is longer than almost all inter-request gaps of the foreground request stream, independent of the intensity. This allows immediate back-to-back execution of foreground requests. While foreground throughput and latencies are very close to 100%, background throughput is almost zero, and its corresponding latency extremely high (factors in the hundreds).

Note that at low foreground intensities of less than 20%, some background requests continue to receive service at preemption intervals of 100-120ms. This is due to an implementation limitation of the FreeBSD *usleep* system call used by the benchmark processes. It can cause sleep intervals to lengthen under high system load, leading to slightly longer inter-arrival times that cause processing of extra idletime requests.

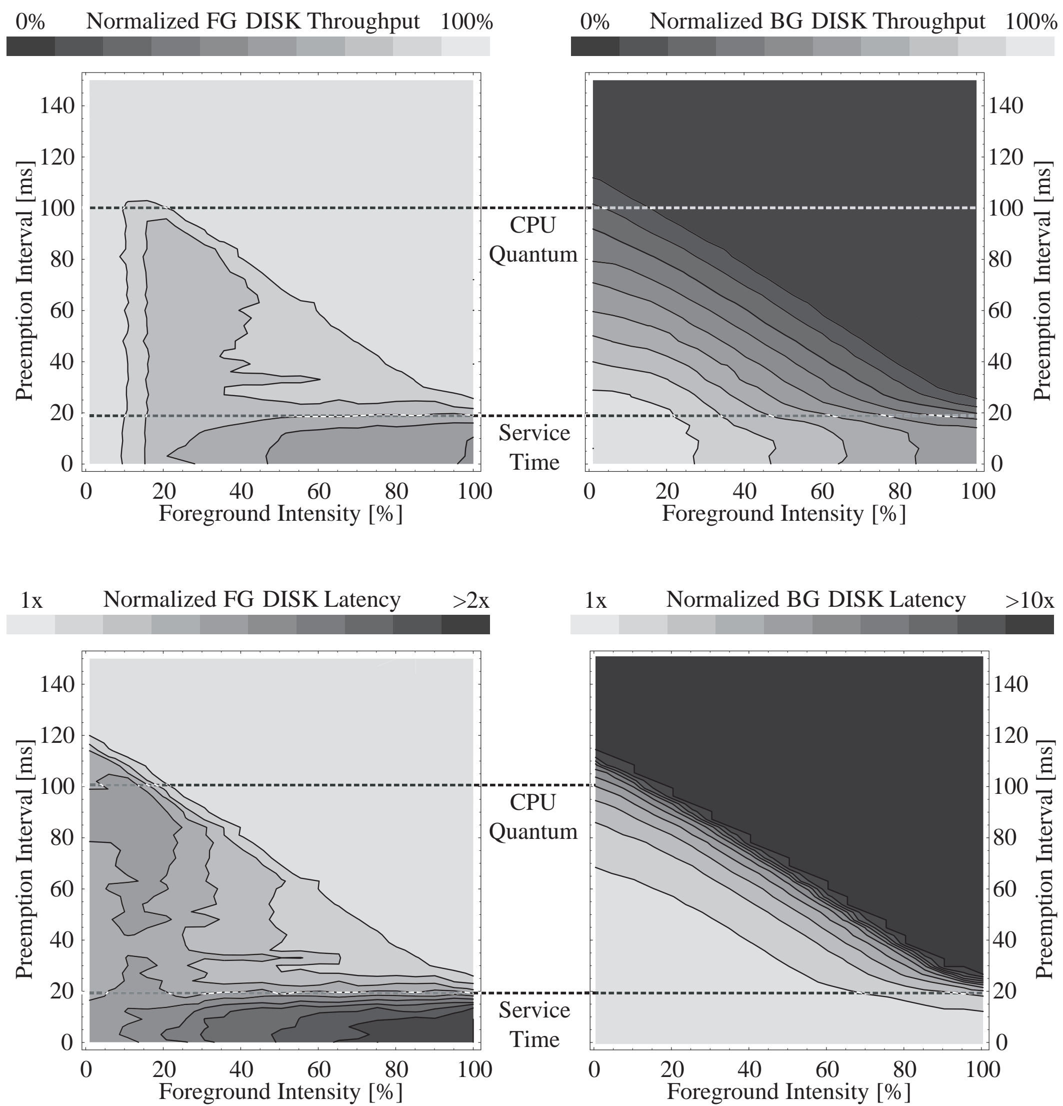


Figure 6.4. Measured random-access disk throughput (top row) and latency (bottom row).

The third region of interest consists of preemption intervals between 20-100ms. Here, idletime scheduling improves foreground performance compared to shorter preemption intervals, but does not completely suppress background processing, as longer preemption intervals do.

For foreground requests, throughput lies between 70-100% with a latency of 1-1.5 times baseline. As preemption interval length increases, idletime use stalls at lower foreground intensities. At a preemption interval of 40ms, background requests are not processed past 90% foreground intensity, whereas with a 80ms preemption interval, this happens at 40% intensity. Section 4.4 discussed how the higher arrival rate of foreground requests lowers the possibility of preemption interval expiration, and thus more easily preempts idletime use.

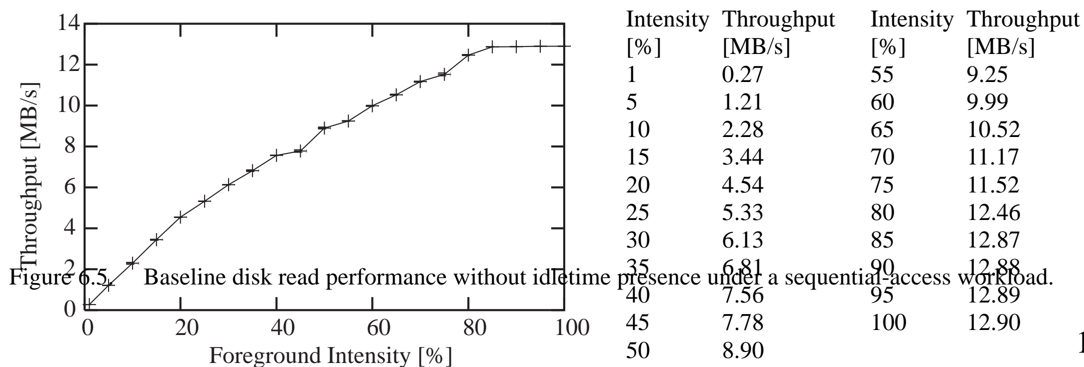
Another observation is that the idletime scheduler is effective at low intensities (less than 10%), no matter what the length of the preemption interval is. This effect is due to priority queuing. With a queue full of background requests, an incoming foreground request always moves to the head of the queue and receives service next. At low intensities, this is sufficient to achieve throughput comparable to the baseline case. However, it must be noted that latency is higher, because queued foreground requests block until the active background request finishes. As foreground intensity increases, priority queuing alone is not capable of maintaining throughput comparable to the baseline, due to the impact of aggregate blockage delays.

In summary, idletime scheduling in this scenario is effective, but may permit a substantial degradation of foreground performance in some cases. This may still be acceptable at lower foreground intensities. A longer preemption interval preempts idletime use at lower foreground intensities, and can consequently help to eliminate the performance impact when workloads demand it.

6.1.2 Sequential Access

The previous section investigated the behavior of the idletime scheduler under a random disk access pattern. This section looks at the sequential case, where both foreground and background processes perform sequential reads of 512-byte blocks.

In the previous random-access scenario, caching of disk blocks was not an issue. Because of the random access pattern across the 8.sGB test file, the probability of repeatedly overlapping accesses was very low. Under the sequential access pattern used in this experiment, cache effects between foreground and background can become an issue.



Intensity [%]	Throughput [MB/s]	Intensity [%]	Throughput [MB/s]
1	0.27	55	9.25
5	1.21	60	9.99
10	2.28	65	10.52
15	3.44	70	11.17
20	4.54	75	11.52
25	5.33	80	12.46
30	6.13	85	12.87
35	6.81	90	12.88
40	7.56	95	12.89
45	7.78	100	12.90
50	8.90		

Figure 6.5 Baseline disk read performance without idletime presence under a sequential-access workload.

Even though the results of idletime read operations are not cached (see Section 5.3), the results of foreground reads are cached as usual. An idletime request for data that is in the cache due to a previous foreground operation will immediately succeed, and result in both an inflated background performance as well as an artificially low impact on foreground performance. The benchmark processes avoid this effect by starting

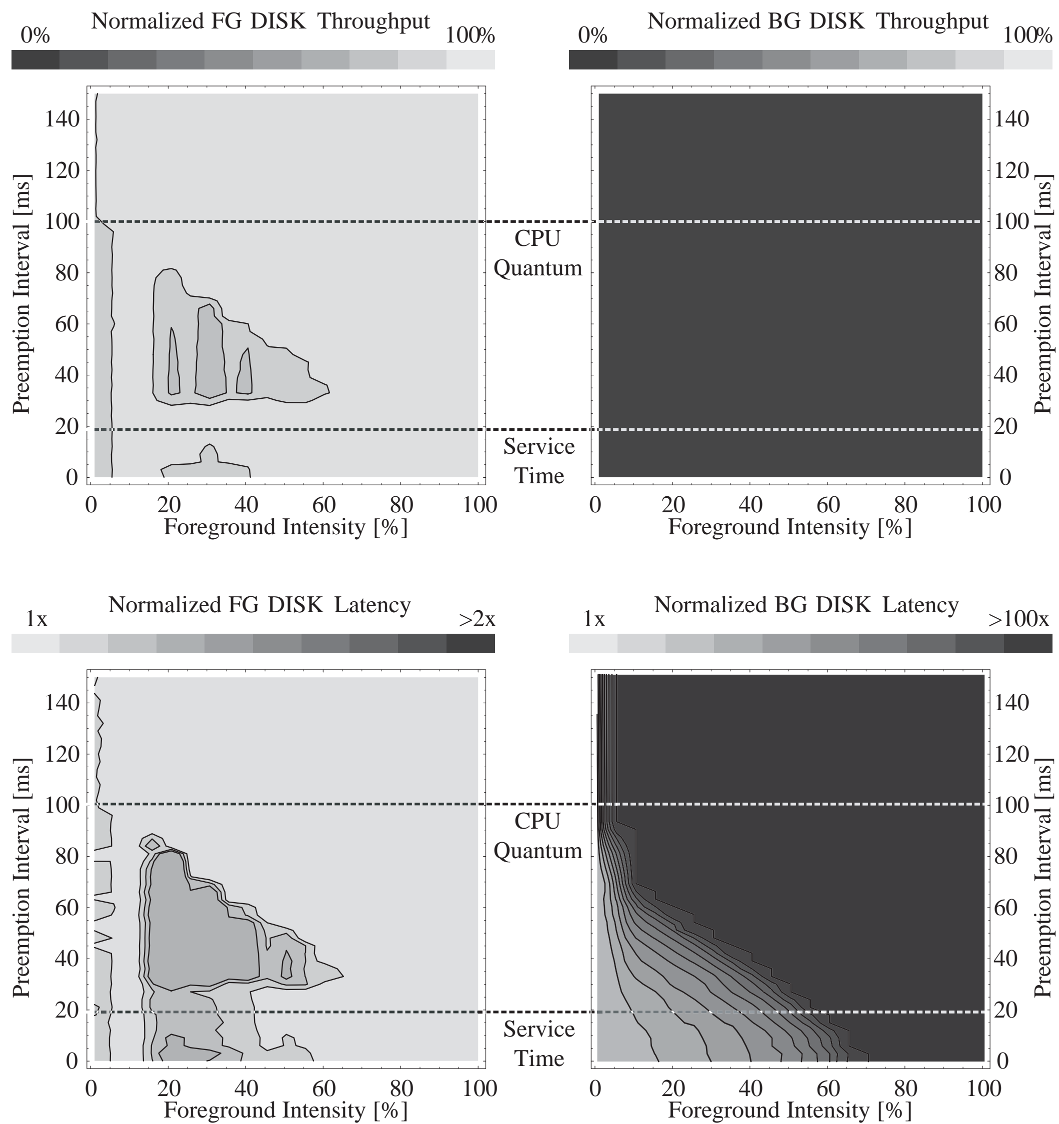


Figure 6.6. Measured sequential-access disk throughput (top row) and latency (bottom row).

their I/O at different offsets into the 8.2GB long test file. The foreground process starts at the beginning, while the background process starts in the middle of the file (4.1GB offset).

During each run, the processes read up to 800,000 blocks at up to 13MB/s throughput. Figure 6.5 shows the detailed baseline performance without idletime presence used for normalization. The error bars indicate standard deviation, which is below 0.07MB/s for all intensities.

The performance graphs for the sequential case in Figure 6.6 are different from the random-access case shown previously in Figure 6.4. The results for this scenario do not exhibit the three distinct areas of behavior found in the previous case.

Sequential foreground throughput is high (80-100%) and latency low (<1.2x) across the board. A smaller, triangular area between foreground intensities of 20-60% and preemption interval lengths of 25-80ms shows a slightly lower foreground performance with throughputs of 70-90% of the baseline and latencies of up to 1.5x the baseline. The random access case (Figure 6.4) has a similar but much more pronounced triangle.

The key difference compared to the random-access case is that in the case of sequential reads, almost no idletime requests receive service. The preemption interval length has very little influence on this effect. Background throughput is between 0-

10% of the baseline case, and corresponding latencies are tens to hundreds of times higher than the baseline.

This starvation of background I/O is due to foreground read-aheads. Section 5.3 explained that in order to avoid foreground delays, the prototype implementation disables read-ahead for idletime operations. However, the system will still issue speculative read-aheads when it detects sequential foreground access patterns. Read-aheads essentially multiply the foreground load, because each read request issued by an application causes the creation of several additional, speculative read operations by the kernel. This occurs even at low intensities, because the kernel heuristic to identify sequential reads solely focuses on spatial locality of successive requests, not their timing.

Disabling read-aheads for foreground disk accesses would address this issue, but would also result in significantly lower foreground throughputs. Because the major focus of the idletime scheduler lies on preserving performance under idletime load, disabling optimizations that benefit foreground performance is not useful.

However, Section 2.2 has discussed how pushing speculative tasks into idletime capacity can improve user-perceived system performance. Section 7.2 will outline future kernel modifications to address this issue, which will execute speculative read-aheads using idletime capacity. A modified read-ahead mechanism may be successful

in maintaining high foreground performance while permitting limited concurrent idletime processing.

6.1.3 Discussion

Section 2.1 explained that the two main criteria for an idletime scheduler are minimal impact on foreground processing and effective utilization of idle capacity. The previous measurements showed that the prototype disk scheduler sustained foreground throughputs of 70-100% of the baseline case under idletime load, with comparable latencies. With high enough foreground intensities, the scheduler completely preempts idletime use, and throughputs and latencies are practically identical to the baseline case.

A main reason for the high impact on foreground performance lies in timing granularities. The disk device used for the experiments has a mean service time of approximately 20ms for random requests. In the random-access scenario, this means that the disk can only serve approximately five random access requests per 100ms CPU quantum. Thus, whenever the idletime scheduler starts servicing a single background request, it may affect foreground performance by up to 20%. This occurs, because the benchmark will generate at least one foreground request per CPU quantum.

The same effect also explains the foreground performance drop using sequential reads in the triangular area with foreground intensities between 20-60% and preemption

intervals between 30-90ms. Even though idletime use receives almost no service, regular performance is still noticeably reduced (10-15%). This is because foreground and background processes read at different disk offsets. Each time the idletime scheduler services a single background request, it still incurs a 20ms seek time, to position the disk head for the background operation. When new sequential foreground reads occur during this time, they must wait for this operation to finish, plus approximately 20ms to move the disk head back. The net result is a potential 40ms foreground delay when an idletime request receives service.

Consequently, the disk benchmark scenario violates the heuristic for determining preemption interval lengths described in Section 4.3.1. The preemption interval should be at least an order of magnitude longer than the service time of the resource, in order to allow amortization of preemption costs across a burst of foreground requests. It should also be significantly less than the inter-arrival time of foreground requests, to allow utilization of some idle capacities for background processing. Because the primary objective of idletime scheduling is isolation of foreground processing from the presence of idletime use, the first rule takes priority whenever a conflict exists.

The disk benchmark scenario does not satisfy both rules. The service time of the resource is 20ms, but preemption interval lengths are less than 150ms. Furthermore, foreground arrival rates even at lightest intensities reach 8-10 requests per second, corresponding to an inter-request gap of only approximately 80ms. According to the heuristic in Section 4.3.1, given the arrival pattern in relation to the service time, the

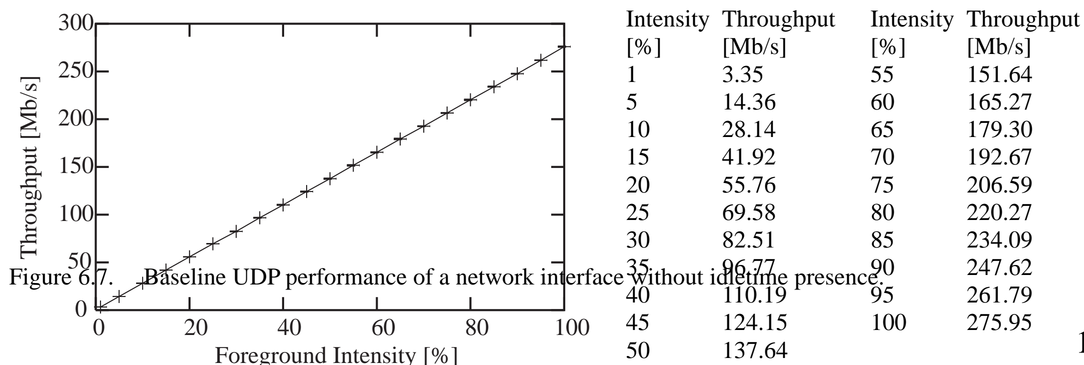
scheduler in this scenario should use a long preemption interval (~200ms) and preempt background processing to prevent interference with foreground use.

6.2 Network Scheduler LAN Evaluation

The previous section presented experimental measurements of the idletime disk prototype, and analyzed them. This section will present a similar discussion for the idletime network prototype.

Crossover patch cords established an isolated, directly connected link between two machines, using Intel PRO/1000F Fiber Gigabit Ethernet interfaces. The interface cards support 64bit PCI-X at 66 MHz, however, the benchmark hosts only featured regular 32-bit PCI slots at 33 MHz. One machine acted as the traffic source, sending a mix of foreground and background traffic towards the sink machine. Each set of experiments evaluates a combination of two different network protocols (UDP and TCP) for foreground and background traffic, resulting in four different experiments to evaluate all protocol combinations.

The TCP benchmark process at the source opens three separate, parallel connections to



Intensity [%]	Throughput [Mb/s]	Intensity [%]	Throughput [Mb/s]
1	3.35	55	151.64
5	14.36	60	165.27
10	28.14	65	179.30
15	41.92	70	192.67
20	55.76	75	206.59
25	69.58	80	220.27
30	82.51	85	234.09
35	96.77	90	247.62
40	110.19	95	261.79
45	124.15	100	275.95
50	137.64		

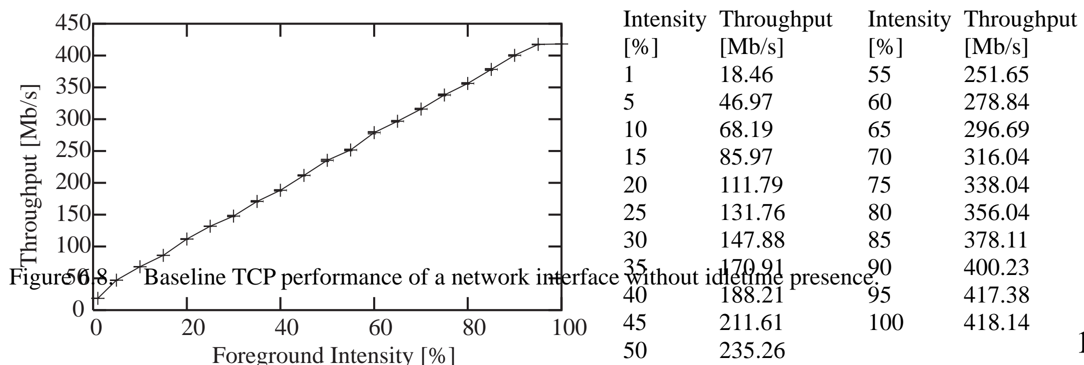
Figure 6.7. Baseline UDP performance of a network interface without idletime presence.

the *discard* service [POSTEL1983] on the sink. The bandwidth-delay-product of a Gigabit link with 1ms delay – which is well above the propagation delay of a local link – is approximately 128KB.

In order to eliminate the socket buffers or system calls as potential bottlenecks, the benchmark process increases socket buffers to 128KB, and then proceeds to send 128K chunks of random data to the sink [JACOBSON1992]. Likewise, the *inetd* process implementing the *discard* service on the sink machine increases its socket buffers to 128KB.

Similarly, the UDP benchmark process uses three separate sockets to send 1400 bytes of random data to the *discard* service on the sink. This avoids fragmentation – the underlying device MTU is 1500 bytes – while allowing experimentation with an early variant of the idletime scheduler that used IP options instead of overloading the type-of-service field.

Unlike TCP send operations, UDP send operations do not block for completion, but instead return an error value if a message was not sent, usually due to outbound queue exhaustion. In such a case, the process sleeps for 10-15ms, allowing the queue to drain



Intensity [%]	Throughput [Mb/s]	Intensity [%]	Throughput [Mb/s]
1	18.46	55	251.65
5	46.97	60	278.84
10	68.19	65	296.69
15	85.97	70	316.04
20	111.79	75	338.04
25	131.76	80	356.04
30	147.88	85	378.11
35	170.91	90	400.23
40	188.21	95	417.38
45	211.61	100	418.14
50	235.26		

Figure 6.8: Baseline TCP performance of a network interface without idletime presence.

before sending more data.

Four separate experiments are required to cover all possible combinations of TCP and UDP foreground and background benchmarks. The preemption interval length for a given run was in effect on both source and sink hosts. Although it has no effect for UDP senders, receiver-side preemption intervals enable correct idletime scheduling of the TCP acknowledgement stream flowing from the sink to the source.

Figure 6.7 and Figure 6.8 show the detailed baseline performances without idletime presence used for normalization. Figure 6.7 displays the UDP baseline performance with error bars indicating the standard deviation, which is below 0.9Mb/s for all intensities. Figure 6.8 shows the TCP baseline performance; the standard deviation in this case is below 1.9Mb/s in all cases.

6.2.1 UDP Foreground Traffic

The first two scenarios transmit foreground traffic via UDP, and use UDP or TCP for idletime transmissions.

6.2.1.1 Foreground UDP vs. Background UDP

Figure 6.9 shows the throughput (top) and latency (bottom) of the foreground (left) and background (right) benchmarks. Both foreground and background benchmarks transmit traffic via UDP. The graphs break down in two major areas based on the length of the preemption interval: less than 0.05ms, and greater than 0.50ms.

With a preemption interval less than 0.05ms, the idletime scheduler is not effective.

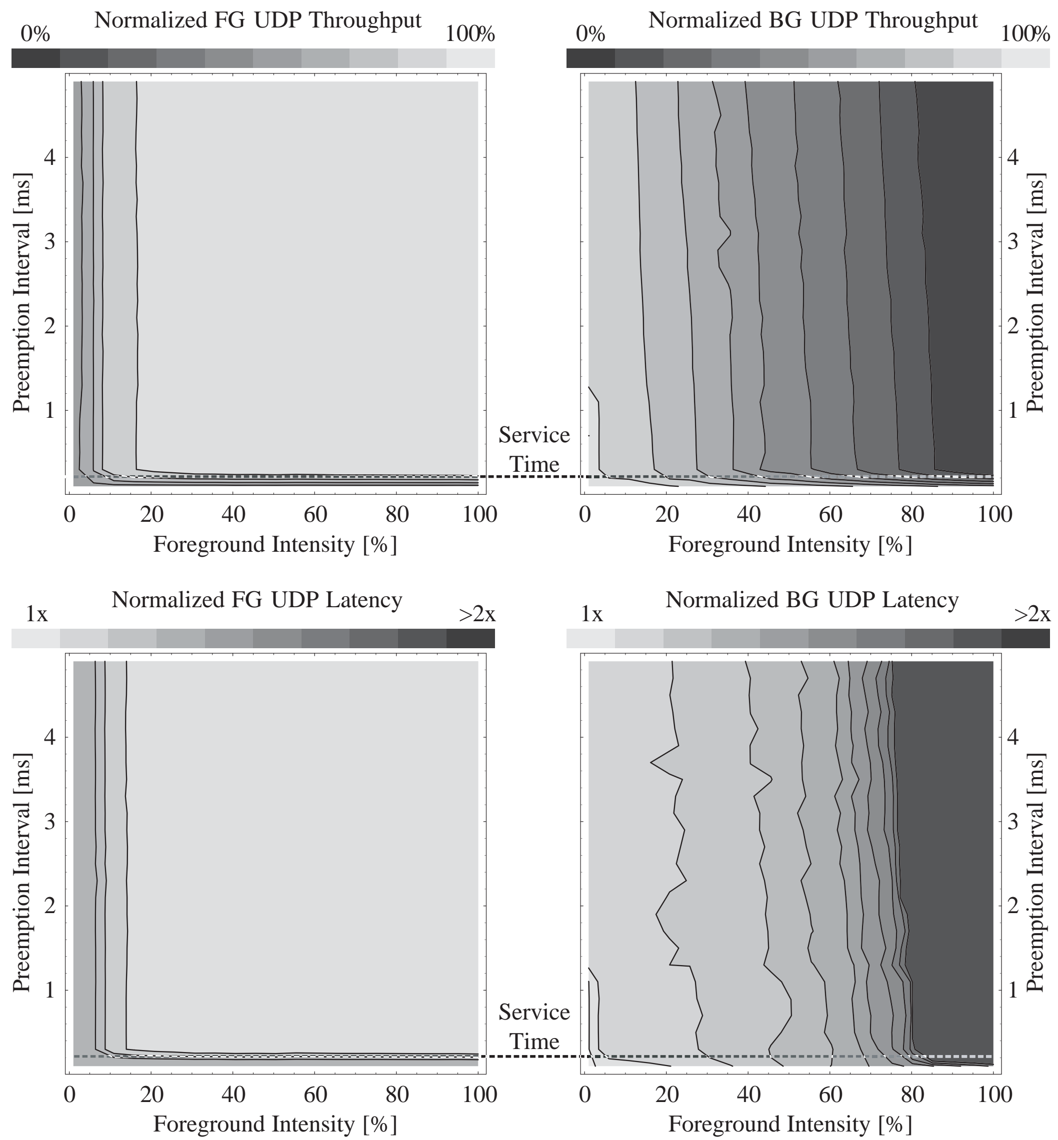


Figure 6.9. Measured 1Gb/s Ethernet UDP/UDP throughput (top row) and latency (bottom row).

Foreground throughput is 50% of the baseline and latency is higher by a factor of 1.4. The background traffic can monopolize the link at lower foreground intensities, and performance of both traffic classes evens out as foreground intensity reaches 100%. This is expected, as the service time of the resource is approximately 0.05ms, including kernel processing.

With a preemption interval longer than 0.05ms, idletime scheduling becomes very effective. At foreground intensities over 10%, both foreground throughput and latency achieve over 90% of the baseline case. With lower foreground intensities, foreground performance still reaches 60-80% of the baseline case. Unlike during the disk measurements, the idletime scheduler here does not suppress background processing completely to maintain unchanged foreground performance. Instead, it gradually reduces the amount of background traffic as foreground intensity increases. Background traffic only stalls at very high foreground intensities (> 90%).

6.2.1.2 Foreground UDP vs. Background TCP

The setup of the second scenario is identical to the first, except that background transmissions use TCP instead of UDP. Figure 6.10 illustrates the measured performance.

Foreground throughput and latency is almost unchanged from the previous case Figure 6.9 with UDP background transmissions. Foreground traffic reaches throughput numbers of 90-100% with latencies close to the baseline case (1-1.1x) at intensities

higher than 20%. For lower intensities, the decrease in performance is slightly higher (70-90% throughput, 1.2-1.4x latency); this is also similar to the case previously discussed.

The scheduler is also effective in utilizing idle capacities for background traffic: background traffic only stalls at high foreground intensities over 80%.

One key difference to the previous scenario (Section 6.2.1.1) is the minimum length of the preemption interval required to make the idletime scheduler effective. In the previous case, the minimum preemption interval was 0.05ms. This scenario requires a minimum preemption interval of 0.3-0.5ms.

This difference may be due to difference in transmission behavior between TCP and UDP. Background TCP traffic that is queued inside the kernel at the socket buffer may receive preferred service compared to foreground UDP traffic that is queued inside the sender process.

To verify this hypothesis, UDP processing could be modified to enable in-kernel queuing. However, this additional buffering may break the traditional UDP API, which requires immediate feedback to the application on send errors, and may lead to application incompatibilities.

One of the strengths of an idletime scheduler using a preemption interval is that it can remain effective in such a scenario, by simply increasing the preemption interval. It does not require modifications to other schedulers, some of which – as this one – are implicit to the operating system.

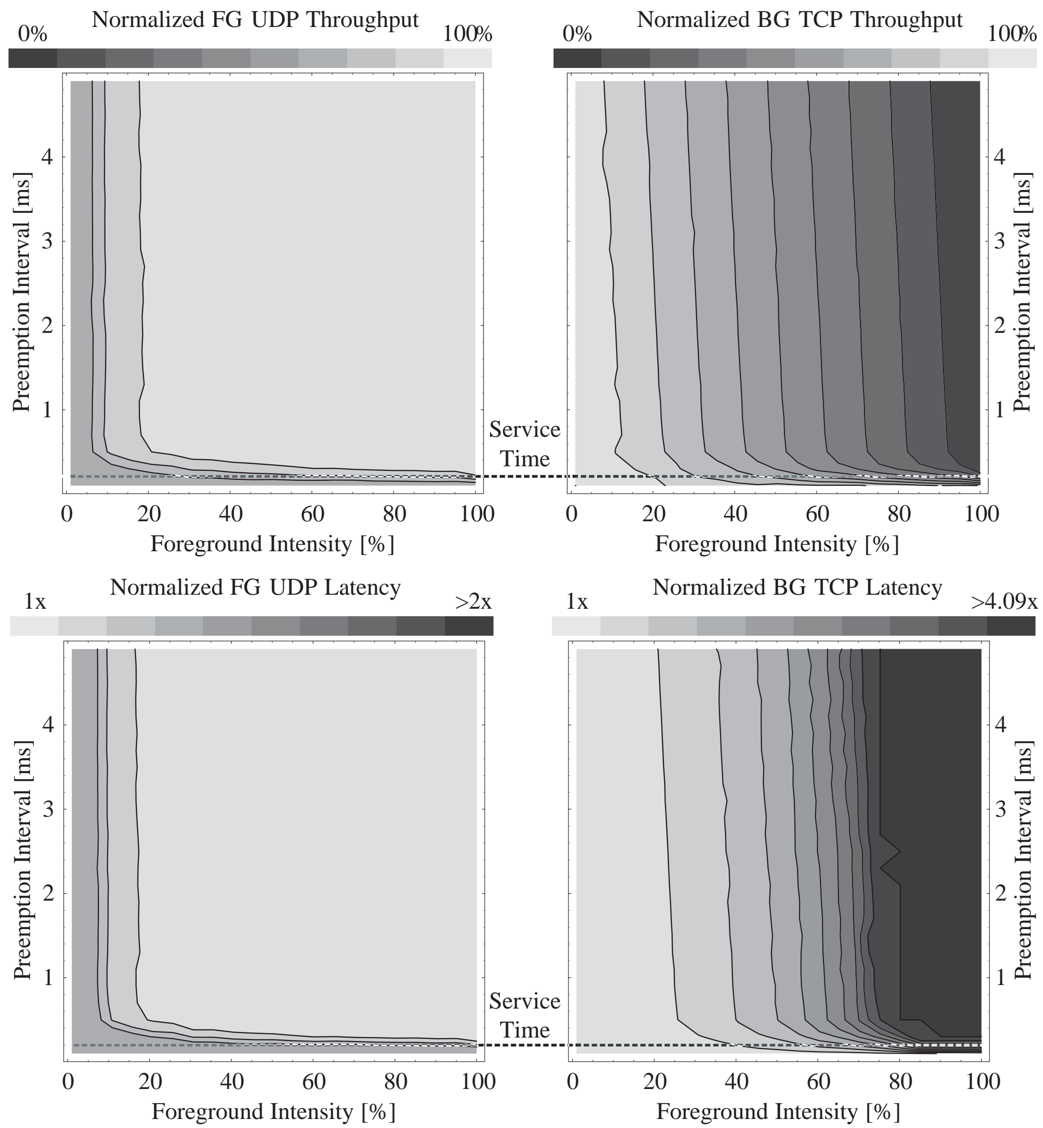


Figure 6.10. Measured 1Gb/s Ethernet UDP/TCP throughput (top row) and latency (bottom row).

6.2.2 TCP Foreground Traffic

The next two scenarios transmit foreground traffic via TCP, and use UDP or TCP for idletime transmissions.

6.2.2.1 Foreground TCP vs. Background UDP

Figure 6.11 shows an experiment where the foreground benchmark uses TCP while the background benchmark uses UDP. In a sense, this represents the worst-case scenario without idletime scheduling: important (hence foreground) congestion-controlled TCP flows share a bottleneck path with greedy, high-rate UDP senders. With FIFO schedulers, the UDP traffic can significantly affect – or even starve –foreground traffic.

In such a scenario, an effective idletime mechanism should still sustain foreground performance at levels that are comparable to the baseline case without background load. Figure 6.11 shows the measured performances in this scenario. Foreground throughput and latency are very close to the baseline case with a preemption interval longer than 1.25ms. Foreground throughput is 90-100% of the baseline, and latencies are 1-1.1x longer. With preemption intervals shorter than 1.25ms, the idletime mechanism is not as effective.

Unlike with foreground UDP traffic, the service time of the resource (0.05ms) is not a useful lower bound for effective service times for TCP foreground traffic.

Significantly longer preemption intervals of 0.9-1.25ms are required to raise foreground performance to levels comparable with the baseline case.

This lower bound of 1.25ms may not be arbitrary. The round-trip time (RTT) estimator in FreeBSD's TCP implementation uses 10ms timers and averages the measurements using fixed-point arithmetic with a scaling factor of eight. This means that 1.25ms is

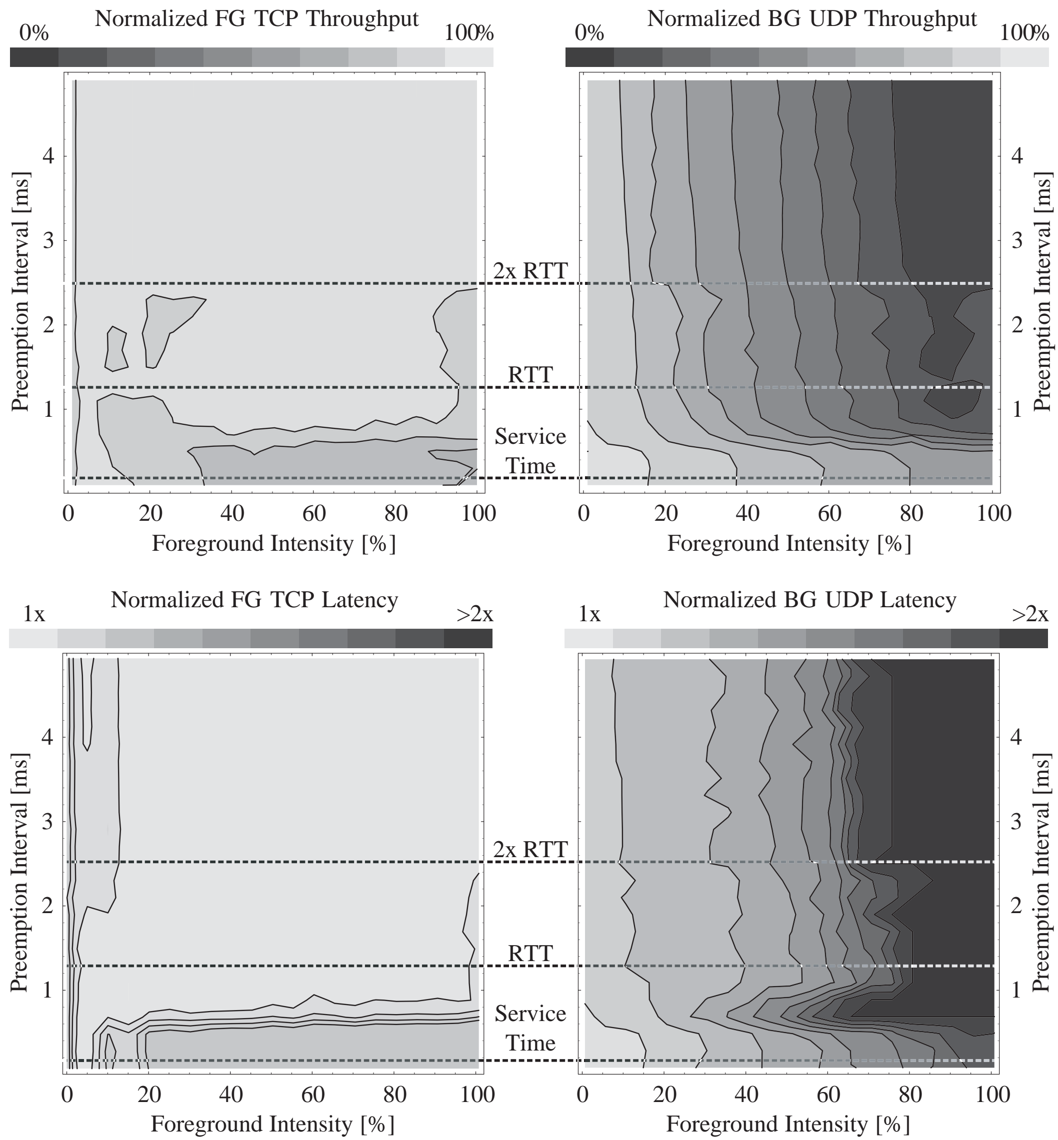


Figure 6.11. Measured 1Gb/s Ethernet TCP/UDP throughput (top row) and latency (bottom row).

the smallest possible RTT estimate for TCP connections. FreeBSD's RTT estimator is a variant of the algorithm described in [BRAKMO1995], which itself is a variant of the original algorithm [JACOBSON1988]. [ARON1998] discusses the effects of the RTT resolution on TCP performance in detail.

A shift in observed performance at approximately 1.25ms could therefore indicate a correlation between effective preemption interval lengths and the estimated RTT of foreground TCP connections. Furthermore, a second, minor performance improvement occurs with preemption intervals of over 2.5ms (twice the RTT). This may indicate a correlation with delayed acknowledgements, which FreeBSD enables by default. Section 6.3 will further investigate these correlations through additional measurements in networks with longer propagation delays.

Figure 6.11 illustrates that the idletime mechanism is again successful in scheduling idletime traffic without interference with foreground transmissions. As in the UDP scenario previously discussed, background use stops only at over 80% intensity. At lower intensities, background throughput reaches up to 80% of the baseline case.

6.2.2.2 Foreground TCP vs. Background TCP

The four graphs in Figure 6.12 show the case where both the foreground and background benchmarks use TCP. As in the previous case, the graphs split into two main regions based on preemption interval length: less than the minimum RTT of 1.25ms and greater than 1.25ms.

The idletime scheduler is ineffective with preemption interval lengths less than 1.25ms and foreground performance is substantially worse compared to the baseline. With preemption intervals longer than 1.25ms, the idletime scheduler becomes effective, and foreground performance is very close to the baseline at 90-100% throughput and 1-1.2x latency.

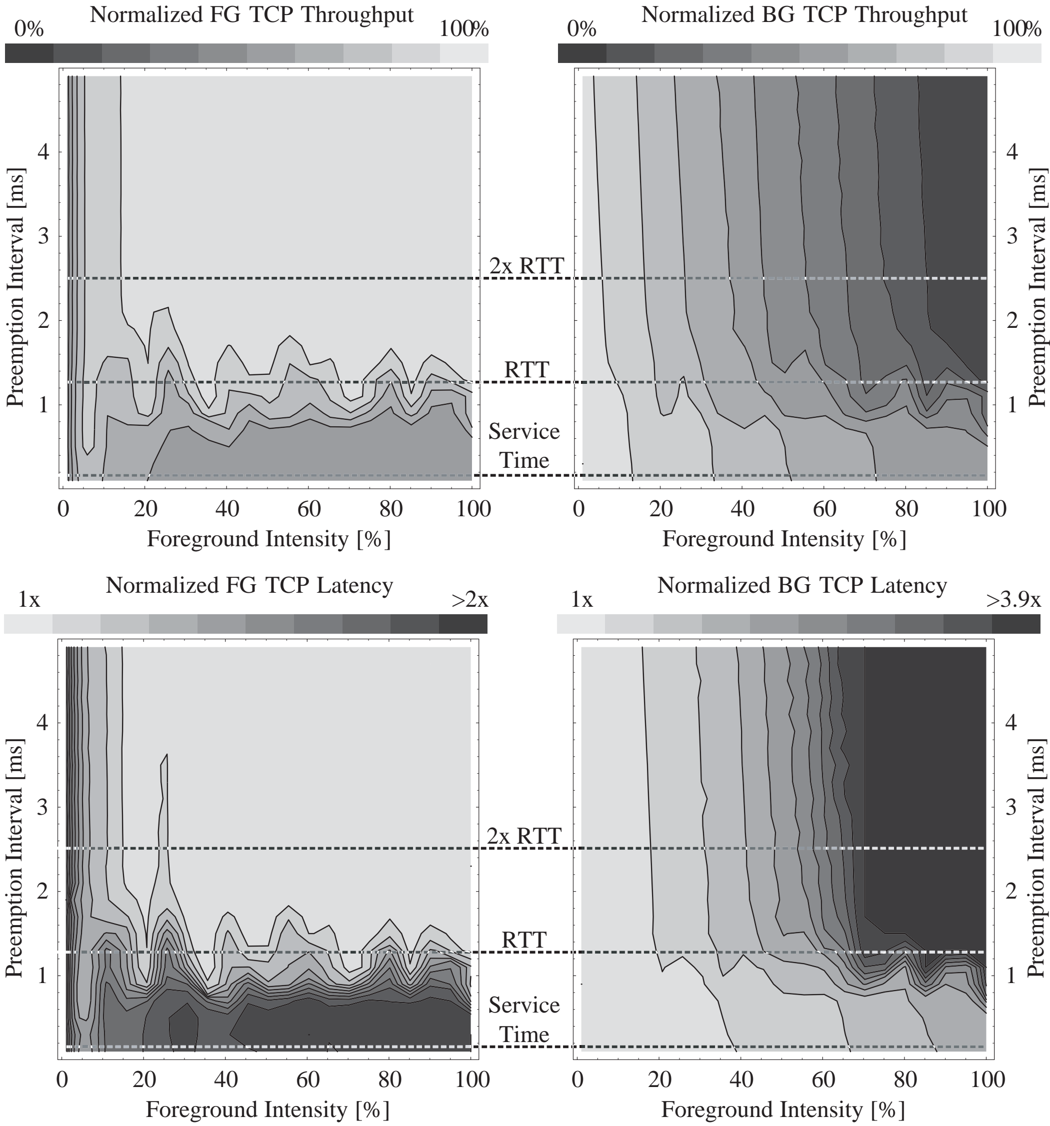


Figure 6.12. Measured 1Gb/s Ethernet TCP/TCP throughput (top row) and latency (bottom row).

The idletime mechanism is again successful in using available capacity to transmit idletime traffic. Only at high foreground intensities (> 80%) do background throughput and latencies decrease, as is required to minimize interference with regular use.

6.3 Network Scheduler WAN Evaluation

When foreground transmissions use TCP, the results presented in the previous section have suggested that the round-trip time of the connection may influence the lower bound for effective preemption intervals.

All previous experiments occurred on a local network (LAN) with small propagation delays less than 0.1ms. FreeBSD's RTT estimator has a maximum resolution of 1.25ms due to internal fixed-point arithmetic. Section 6.2.2 found that in experiments with TCP foreground transmissions, effective preemption intervals must exceed the RTT. Lengthening the preemption intervals past twice the RTT resulted in another, minor performance improvement.

To determine whether the link delay between the benchmark machines affects the idletime mechanism, the experiments with TCP foreground traffic (Section 6.2.2) must be repeated over a similar wide-area (WAN) link with a longer propagation delay.

Dummynet [RIZZO1997] is a FreeBSD kernel mechanism to apply artificial delays, queue limits, and loss rates to selected flows. *Dummynet* can simulate a wide-area link

by buffering packets in a transmission queue – sized to accommodate the bandwidth-delay-product of the chosen link – for a given delay.

However, simulating wide-area Gigabit links with *Dummynet* is problematic [ZEC2003]. *Dummynet* uses the kernel firewall to identify packets for processing, and depends heavily on the kernel timers to control when packets leave the transmission buffer. Both mechanisms incur significant overheads at high data rates. Furthermore, high data rates cause high interrupt loads, which can decrease system responsiveness and eventually lead to livelock [MOGUL1997]. Because *Dummynet* processing occurs at the IP layer, device interrupts cause delays that reduce the accuracy of the simulation. These delays can also interfere with user-space processing, and as a result affect the benchmark processes themselves.

Changing the experimental topology from a direct link to a two-hop connection could eliminate these issues. A fast intermediate router could perform all *Dummynet* processing; the benchmark systems would perform the same processing as in the directly connected case. However, this change in topology modifies the experimental setup in ways that may make comparison between the LAN and WAN scenarios invalid.

Switching to a slower link speed is another approach to eliminate *Dummynet* performance issues. A short, empirical investigation has shown that the benchmark systems used in the previous sections are powerful enough to simulate a 100Mb/s

Ethernet link at 10ms delay using *Dummynet*. The speed of the underlying link should not affect whether the RTT has an impact on effective preemption delays for TCP foreground traffic.

The wide-area experiments presented in the remainder of this section thus retained the one-hop topology used in the Gigabit experiments, and instead replaced the Gigabit link with a slower 100Mb/s Ethernet connection. Two additional experiments were run. The first one, in Section 6.3.1, repeats the Gigabit experiments that used TCP to transmit foreground traffic (Section 6.2.2) over the local, directly connected 100Mb/s link. Again, a crossover cable connected the test machines, using two Intel PRO/100 Fast Ethernet adaptors. This first set of experiments establishes a baseline to verify that the idletime scheduler is effective in a 100Mb/s LAN.

The second experiment, in Section 6.3.2, repeats this setup, but increases the link delay to 10ms using *Dummynet*. A buffer of 128KB simulates the bandwidth-delay-product of the link. The increase in delay and buffer size simulates a WAN link, and allows investigation of whether the propagation delay of the link influences the range of effective preemption intervals.

6.3.1 100Mb/s Baseline

This section repeats the experiment from Section 6.2.2 over the 100Mb/s LAN link. The experiments run without artificial delays, to establish a baseline for the next of experiments. As before, the propagation delay of the underlying link is less than the

resolution of TCP's RTT estimator. As a result, TCP measures the RTT in this setup at 1.25ms.

Figure 6.13 shows the measurements when the foreground traffic is transmitted over TCP and background transmissions use UDP. As for the corresponding Gigabit case (Section 6.2.2.1), the idletime scheduler requires preemption intervals longer than the

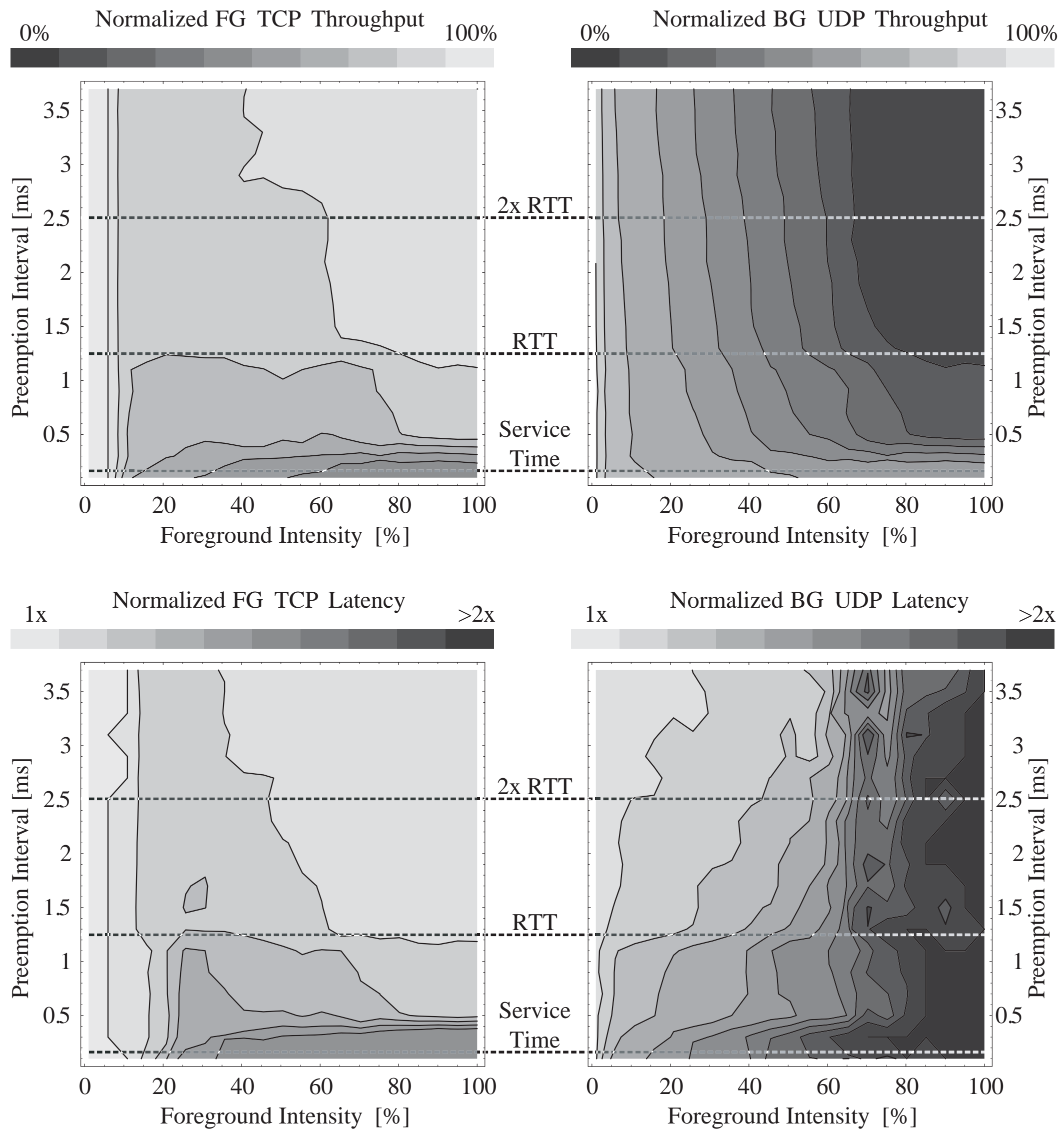


Figure 6.13. Measured 100Mb/s Ethernet TCP/UDP throughput (top row) and latency (bottom row).

RTT of 1.25ms for effective operation. Increasing the preemption interval over twice the RTT results in another, minor performance improvement.

Figure 6.14 shows the next scenario, where both foreground and background traffic use TCP. This corresponds to the Gigabit case examined in Section 6.2.2.2. Again, with preemption intervals above the RTT, the idletime mechanism is more effective in

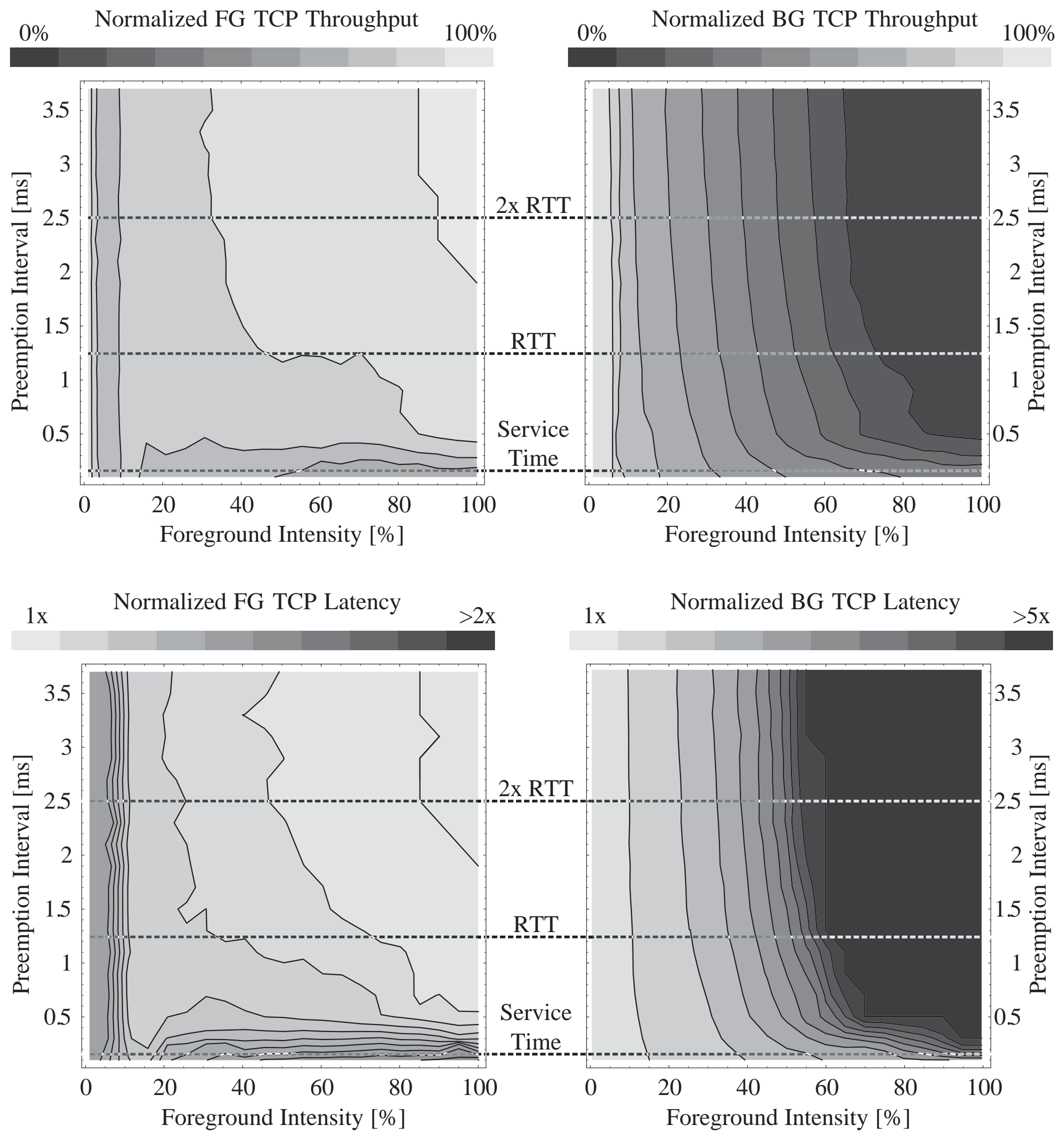


Figure 6.14. Measured 100Mb/s Ethernet TCP/TCP throughput (top row) and latency (bottom row).

isolating foreground traffic from the presence of idletime use.

One difference to the previous case with UDP background traffic (Figure 6.13) is that raising the preemption interval past twice the RTT has no significant effect.

As in the Gigabit scenarios presented in Section 6.2.2, the RTT influences the range of effective preemption intervals in the 100Mb/s LAN case.

6.3.2 100Mb/s with 10ms Delay

The results in the previous section have indicated that TCP's RTT estimate influences the range of effective preemption interval lengths for a LAN link. This section modifies the previous setup. It uses *Dummynet* to simulate a 100Mb/s WAN link with 10ms delay, as described at the beginning of Section 6.3.

Figure 6.15 shows the first scenario, where TCP foreground traffic competes with UDP background traffic on the simulated WAN link. The estimated RTT of the foreground connections clearly affects the minimum effective preemption length. In the corresponding LAN case (Section 6.3.1, Figure 6.13) the idletime scheduler became effective with preemption intervals longer than the corresponding RTT of 1.25ms in the LAN scenario. Here, in the WAN case with 10ms delay, the required preemption length for effective idletime scheduling is over 20-30ms.

With a preemption interval less than 20-30ms, foreground throughput reaches only approximately 50% of the baseline, and the corresponding latencies range from over

2x to 1.5x of the baseline. Likewise, with preemption intervals less than 20-30ms, the background throughput is high (70-90% of the baseline) and the corresponding latencies are very low (1-1.3x baseline).

It is interesting to note that effective preemption lengths of 20-30ms are slightly longer than the simulated RTT of 20ms. Two factors may contribute to this effect. First,

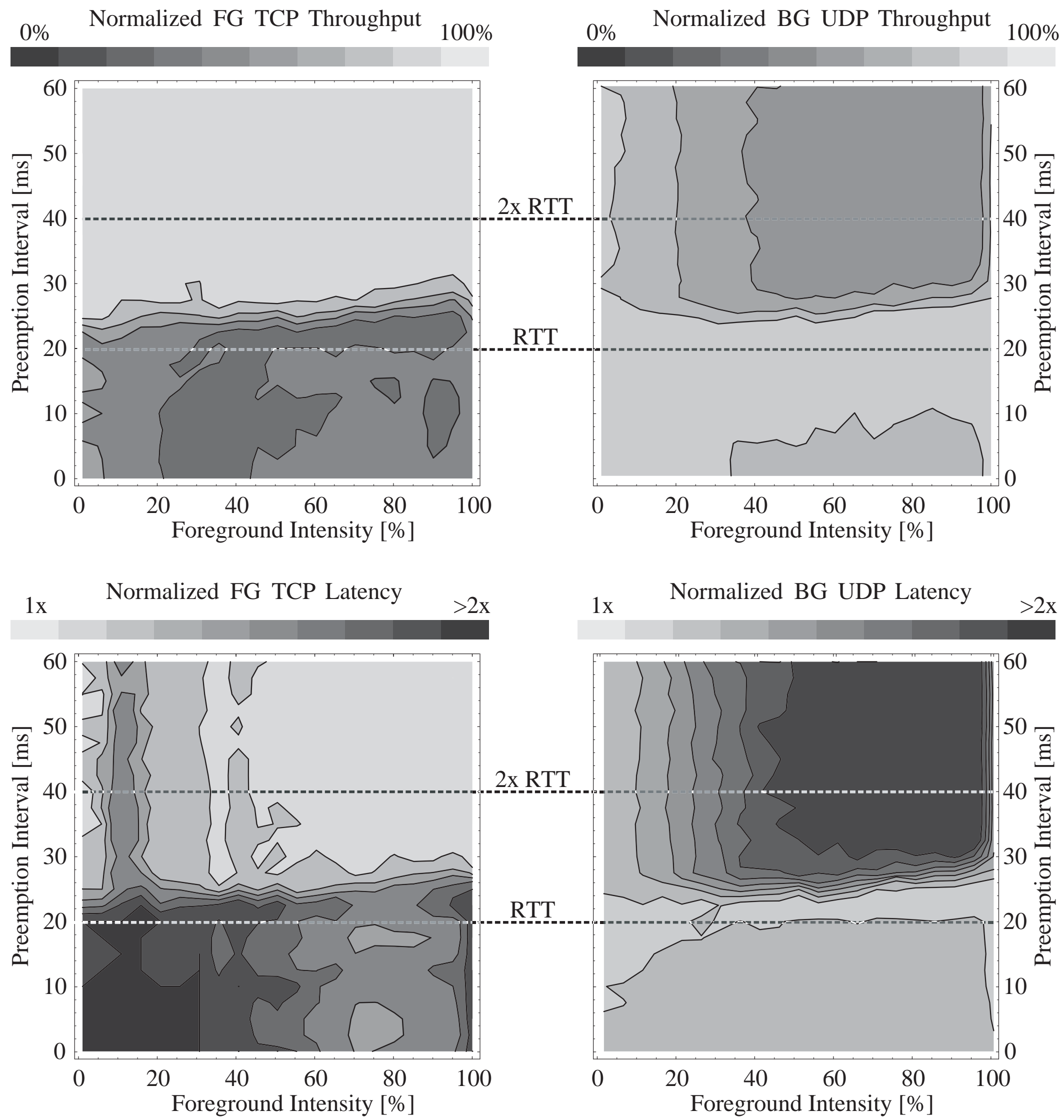


Figure 6.15. Measured 100Mb/s Ethernet TCP/UDP throughput (top row) and latency (bottom row) with 10ms delay.

TCP's RTT estimator conservative by design and over-estimates the RTT to avoid overloading the network. Second, the link is fully loaded during these measurements, because background transmissions are greedy. Additional queuing delays can therefore increase the apparent RTT.

Figure 6.16 shows the scenario where both foreground and background transmissions use TCP. Again, the minimum preemption interval at which the idletime scheduler becomes effective lies between 20-30ms.

One large difference between this scenario and the corresponding LAN case without additional delays (Section 6.3.1, Figure 6.14) is background throughput. In the LAN case, background throughput reached up to 50% under light foreground intensities, with preemption intervals longer than the RTT, and thus without affecting foreground transmissions. Here, background transmission stops almost completely as the preemption interval exceeds than the RTT. Additional experiments are required to investigate this phenomenon. One hypothesis is that TCP's global timers synchronize transmission processing for different connections. This causes foreground and background packets to compete for transmission after a timer fires, and results in amplified background delays due to idletime scheduling. The decrease in background performance with preemption intervals shorter than the RTT, compared to the UDP scenario, would support this hypothesis.

Another possibility is that these results are due to the network environment experienced by background TCP connections. Due to prioritizations, the network and end systems will drop background packets first during times of congestion. Furthermore, preemption intervals cause frequent stalls in both background data and acknowledgment streams. These stalls could force background connections into

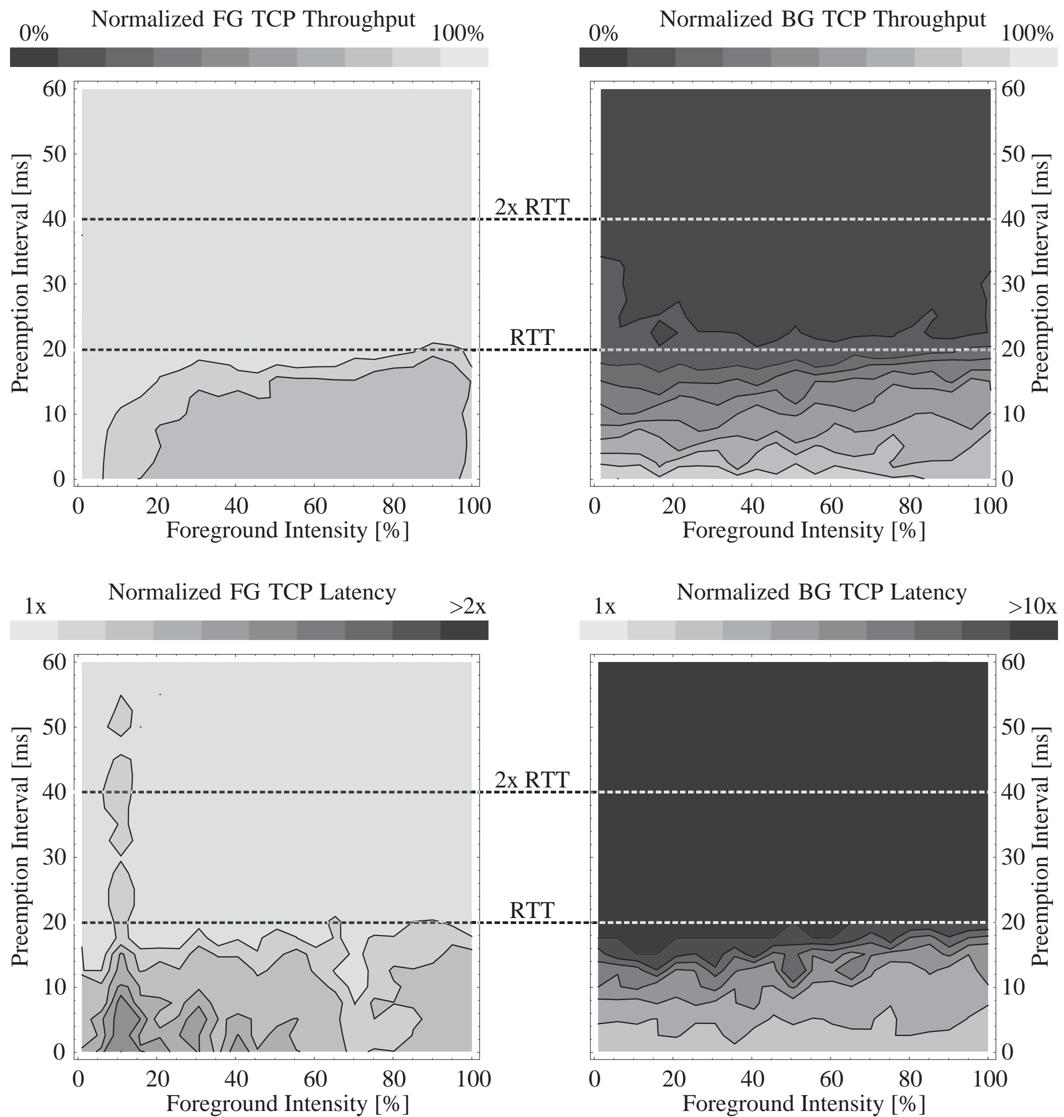


Figure 6.16. Measured 100Mb/s Ethernet TCP/TCP throughput (top row) and latency (bottom row) with 10ms delay.

repeated slow-start as foreground load and preemption interval length increase. Section 7.1.4 discusses this in detail.

Additional, detailed experiments are required to understand this behavior fully. However, because idletime scheduling isolates foreground traffic from the presence of background use, using TCP for background transmissions serves no purpose. Congestion-uncontrolled UDP traffic will optimize background throughput without decreasing foreground throughputs.

The main purpose of the experiments presented in this section was to investigate a possible relationship between the RTT and the minimum required preemption interval for effective idletime scheduling. The measurements on both Gigabit and 100Mb/s LAN links have exhibited a significant foreground performance increase as preemption interval lengths exceeded TCP's RTT estimate of 1.25ms. The measurements presented in this section for a simulated 100Mb/s WAN link with 10ms delay have confirmed this relationship. Additional measurements are required to investigate the behavior of the mechanism over different link speeds and delays, but similar behavior would be expected.

6.4 Network Scheduler Discussion

In the disk case discussed in Section 6.1, the ratio of foreground arrival rate to resource service time was such that the idletime scheduler had to preempt background

use completely to prevent a significant decrease in foreground performance. This is not the case for network interfaces, where service times are approximately 0.05ms.

In the Gigabit LAN scenarios (Section 6.2), the idletime scheduler is very effective. It sustains foreground throughputs higher than 90% of the baseline, with comparably low latencies. Furthermore, the scheduler utilizes available idle capacities for background load until foreground intensity reaches approximately 80%. At higher intensities, it preempts idletime use to protect foreground performance.

The experiments illustrated other interesting points. First, as expected, foreground performance at low intensities (<10%) can become affected by the presence of idletime service, because the burst length at low intensities is short. Because idletime scheduling amortizes preemption cost over a foreground burst, this results in a higher per-request overhead, leading to an overall reduction in performance. Section 7.2.1 will further examine this case.

Second, the minimum preemption interval length when TCP is used to send foreground traffic (Sections 6.2.2 and 6.2.2.1) is approximately 1.25ms and therefore significantly higher than with UDP at 0.05ms. Because 1.25ms is the resolution of TCP's RTT estimator, a possible correlation between estimated RTT and effective preemption delay lengths was investigated in Section 6.3. The measurements over a simulated WAN link with 10ms delay have confirmed that effective preemption interval lengths depend on the estimated RTT of the network link when TCP is used to

transmit foreground traffic. Additional experiments are required to determine the exact nature of this relation.

6.5 Experimental Limitations

The experiments presented in this chapter investigated the behavior of the idletime schedulers in several interesting scenarios. However, further experiments that evaluate the mechanism in additional scenarios will aid in better understanding the benefits and limitations of the proposed scheduler.

First, all network and disk scenarios use unlimited idletime workloads that strive to utilize all available capacity. This models the worst-case scenario for an idletime scheduler, because many foreground requests will incur preemption costs, due to ongoing idletime use. During this initial evaluation of the idletime scheduler, focus on worst-case behavior allowed investigation of the feasibility of idletime scheduling with preemption intervals.

The resulting measured performances consequently are lower and upper bounds for foreground and idletime processing, respectively. With more realistic, limited idletime workloads, foreground performance can increase, due to preemption cost reductions during periods of idletime inactivity.

Another limitation was the use of synthetic foreground workloads. For this initial set of experiments, measuring a wide range of foreground loads was important, to arrive

at an initial evaluation of the overall behavior of the idletime mechanism. In the future, subjecting the scheduler to actual user-generated workloads will allow investigation of the concrete, application-perceived impact on foreground performance.

In addition, the network experiments in Sections 6.2 and 6.3 only investigate a single, directly connected network topology. Some experiments also vary the propagation delay of that link. Experiments with a static topology allow direct performance comparisons between different combinations of transport protocols. However, an important remaining question is behavior of the idletime scheduler in Internet-spanning end-to-end scenarios.

Implementing the proposed scheduler in a network simulator, such as *ns2* [BRESLAU2000], can aid in these evaluations. The *ns2* simulator, and its *nam* visualization system, allows realistic simulation of wide-area Internet paths. It includes standards-compliant implementations of the Internet protocol suite, and can generate realistic traffic patterns that model aggregate user traffic.

6.6 Summary

This chapter evaluated the performance of the prototype implementation described in Chapter 5 experimentally. It has investigated the behavior of the idletime scheduler for two resources with different characteristics: disk drives and network interfaces. For

both types of resources, multiple experiments measured the achieved foreground and background throughputs and latencies over a range of load patterns and preemption interval lengths.

The idletime disk scheduler effectively protected foreground performance under a random-access workload by frequently preempting background use due to timing granularity. With a sequential-access workload, the speculative read-aheads – issued by the system to increase performance – almost completely prevented idletime use. A future version of the disk scheduler should better incorporate the aggressive read-ahead load, and still allow some idletime use.

The idletime network scheduler performed well under both UDP and TCP workloads, and successfully maintained high foreground performance while supporting idletime. Idletime use continued up to foreground intensities of 80%. At higher intensities, the scheduler started to stall transmission of background traffic, to avoid decreasing foreground performance under heavy load. One interesting observation with TCP senders was that the required minimal preemption interval corresponds to the round-trip time instead of the resource service time. Additional experiments are required to investigate this correlation further.

7. Discussion

The previous chapter presented an experimental investigation of the performance of the idletime scheduler under a variety of workloads. This chapter will discuss the conclusions of this evaluation and highlight the main results and issues, such as differences between the predicted and measured scheduler behavior. The second part of this chapter describes proposed future extensions that can improve certain aspects of idletime scheduling, such as automatically adapting the preemption interval length to the observed foreground workload.

7.1 Overview

The results of the experimental in Chapter 6 confirm that an idletime scheduler based on preemption intervals is generally effective in reducing the impact on foreground performance caused by idletime use. Foreground performance decreases less than 10-15% under many worst-case idletime workloads. However, the experiments identified several issues that the theoretical analysis of the mechanism did not predict. The remainder of this section will discuss these issues.

The first part of this section will compare the measurement results to the predicted performance based on the quantitative analysis of Chapter 4. Despite the simple nature of the model, its performance predictions conformed to the measured behavior with an overall error below 15%. The quantitative analysis illustrated how changing the length of the preemption interval allows trading a reduction in foreground performance

against increased idletime performance, or vice versa. Section 7.1.2 discusses observations of these effects during the experiments.

One limitation of the quantitative analysis is the assumption of simple workloads, such as fixed-rate UDP traffic or disk accesses. It cannot predict performances for more complex workloads, such as TCP flows. It consequently failed to model the effects observed during some experiments in Chapter 6, where foreground TCP flows required preemption intervals longer than the RTT to reach acceptable performance. This result, further discussed in Section 7.1.3, does not invalidate the overall mechanism. For such flows, the idletime scheduler still maintains foreground performances comparable to the baseline with preemption interval lengths longer than the RTT.

Background TCP flows can suffer from a different problem. The TCP protocol operates well under stable or slowly changing network conditions, and starts to perform poorly when network conditions fluctuate on smaller timescales. The network environment experienced by background traffic features rapidly changing bandwidths and propagation delays. Consequently, TCP background throughput can be poor. Section 7.1.4 investigates these interactions in detail. Because the idletime scheduler will protect foreground flows from the presence of arbitrary idletime traffic, a more aggressive transport protocol, such as UDP, improves background performance without affecting foreground traffic.

The measured performances of the sequential disk benchmark in Section 6.1.2 were also different from the predictions based on the quantitative model. Kernel-generated read-aheads multiplied the application-generated foreground load, and caused an almost complete stop of idletime processing. Section 7.1.5 discusses this finding, and proposes a modified read-ahead mechanism to allow idletime use while maintaining high foreground performance.

7.1.1 Measured vs. Predicted Performance

Section 4.4 presented a simple quantitative analysis of the model of idletime processing formally defined earlier in Chapter 4. This quantitative analysis predicts the global behavior of the idletime mechanism for a given resource and workload. This section will compare the predicted performances for the network (Section 4.4.3) and disk (Section 4.4.2) scenarios against the measured performances observed during the experimental evaluation in Chapter 6.

Due to the simplifying assumptions during the analysis, the resulting predictor is not expected to describe the performance at very high accuracy. Instead, its purpose is to estimate the global behavior of the mechanism over a wide range of workloads and resources. The absolute errors between predictions and experimentally measured results are less than 15% in most cases, indicating that the prototype implementation conforms to the expected behavior. The next sections compare predictions against measurements for two scenarios in detail.

7.1.1.1 Gigabit LAN

Figure 4.21 in Section 4.4.3 showed the predicted foreground and background throughputs for a local-area Gigabit link. The quantitative analysis in Section 4.4 is simple, and cannot express the behavior of congestion-controlled transport protocols such as TCP. This discussion will hence focus on the scenario where both the foreground and background sender use UDP. The modeled scenario corresponds to the experimental setup measured in Section 6.2.1.1. The top graphs in Figure 6.9 show the observed throughputs during the experiment.

To compare the predictions to the measurements, Figure 7.1 shows the absolute difference between the two sets of numbers, displayed as a contour graph. Note that the shades of the contours are scaled; black areas correspond to a 50% difference between the prediction and the measured performance.

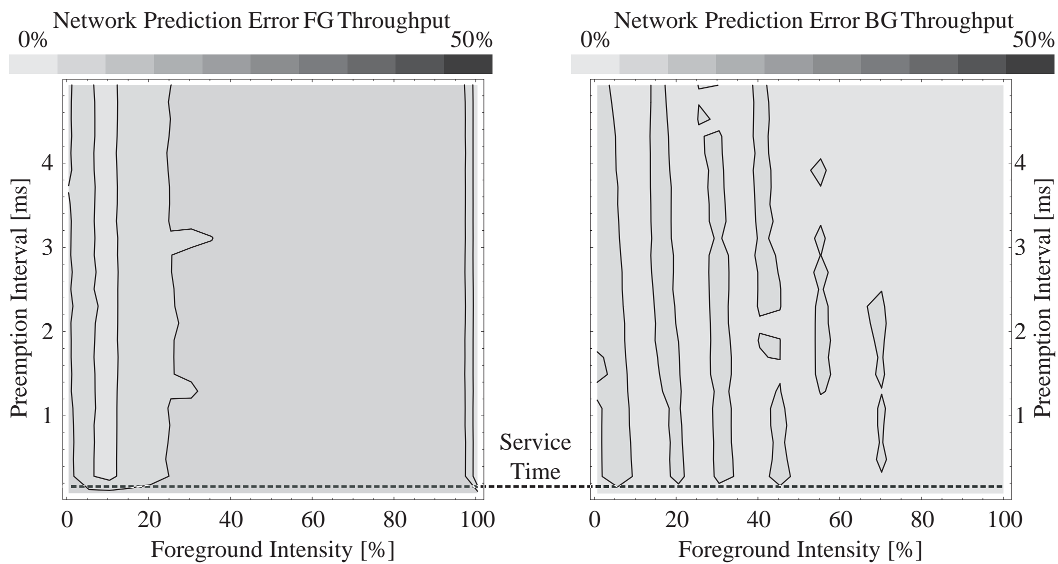


Figure 7.1. Relative performance prediction error for the network case.

Figure 7.1 illustrates that the simple prediction function derived from the model in Section 4.3.1 is effective: maximum overall prediction error is less than 10%. Furthermore, the regions where the predictor performs worst are those where the idletime scheduler itself is ineffective. This occurs mostly at low intensities less than 10%, or when the preemption interval is shorter than the service time (here, 0.05ms). Outside these areas, the performance prediction is more accurate with only a 1-5% error for both foreground and background throughputs.

7.1.1.2 Disk Drive

Section 4.3.1 also predicted performance for a disk resource under a random-access workload. The experimental evaluation measured this case in Section 6.1.1 and showed the measured throughput in the top graphs in Figure 6.4. As in the network performance comparison discussed in the previous section, Figure 7.2 shows the prediction error for this scenario. As in above in Figure 7.1, black regions in Figure 7.2

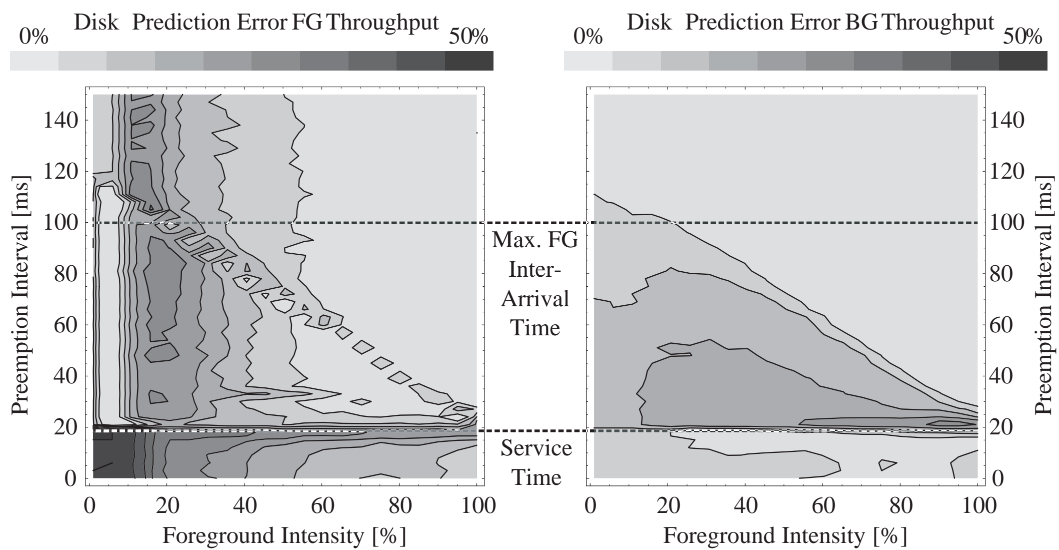


Figure 7.2. Relative performance prediction error for the disk case.

correspond to a 50% prediction error.

As in the Gigabit LAN case, the predictor is ineffective in predicting disk throughputs when the preemption interval is shorter than the service time of 20ms. It performs worst at light intensities and short preemption interval lengths, where the prediction error can reach 40%. The cause of this high error percentage again lies in the simple model underlying the analysis.

With preemption intervals longer than the service time, the performance prediction is much more accurate. It predicts performance with overall errors less than 15%. For light foreground intensities less than 20%, the prediction error can increase by 5%.

In the Gigabit network performance comparison discussed in the previous section, the predictor achieved identical accuracies for both foreground and background throughputs. However, the performance predictor for the disk resource shown in Figure 7.2 predicts background throughput more accurately than foreground throughput. The foreground prediction error can reach 15-20%, while background errors remains less than 10% overall. This may indicate that factors other than the available bandwidth affect foreground performance. One factor not included in the simple quantitative analysis is the variability of disk access costs due to disk head location. This omission may affect the accuracy of the predictor.

A detailed comparison of the measured results against the quantitative model finds that the model does not accurately describe observed performance in borderline cases. For example, the model assumes a linear decay of background performance with increasing preemption interval lengths. A detailed analysis of the measured results indicated that background performance instead exhibits a logistic decay model as preemption interval lengths increase, especially at high foreground loads. This causes the pronounced diagonal area of increased error rates visible in the left image in Figure 7.2.

Overall, however, the quantitative analysis is successful in predicting global behavior of the idletime scheduler for different resources and workloads with acceptable error rates, and serves as an indicator of the correctness of the implementation.

7.1.2 Effects of Preemption Interval Length

The analysis of the idletime scheduler in Chapter 4 identified upper and lower bounds on useful preemption interval lengths. The lower bound is resource-dependent, such as the service time or link RTT. The upper bound depends on the workload, and corresponds to the inter-arrival time of foreground requests. The length of the preemption interval allows tuning the scheduler's aggressiveness in utilizing idle capacity. This property allows adaptation of the mechanism to various resources, workloads, and user policies on acceptable performance impacts. One direction of future research is mechanisms to choose effective preemption interval lengths

automatically, based on workload and user policy. Section 7.2 discusses such approaches.

Idletime scheduling amortizes preemption delays due to idletime use over bursts of foreground requests. Section 4.3.1 defined a foreground burst as a sequence of foreground requests whose starting times lie within the respective predecessor's preemption interval. The idletime scheduler is less effective when these foreground bursts are short. Each burst will at most incur a single preemption. With short bursts, this results in higher reduction of foreground performance. The disk benchmarks in Section 6.1 illustrate this case. Service times of 20ms with foreground arrival rates in the same order of magnitude resulted in short foreground bursts, causing less effective idletime scheduling that did not fully prevent foreground delays.

Short foreground burst lengths also appear when the intensity of the foreground requests is low, as discussed in Section 4.3.1. Consequently, the scheduler cannot completely isolate low-rate foreground workloads from concurrent idletime use. This may be acceptable in some scenarios, because a very light foreground load often indicates that full performance of a resource is not required. In cases where this delay is not acceptable, a longer preemption interval will create longer foreground bursts that reduce overhead at the cost of a substantial reduction in background performance. When performance of low-rate foreground workloads is paramount, a preemption interval longer than the inter-arrival time will halt idletime processing, and guarantee foreground performance comparable to the baseline scenario.

Thus, the inter-arrival time between foreground requests acts as an upper bound for useful preemption interval lengths. With preemption intervals longer than the foreground inter-arrival time, idletime use stops completely, and foreground processing occurs as if no idletime scheduling was present. The disk experiments in Section 6.1 illustrate this behavior. When preemption interval lengths exceed the foreground inter-arrival time of 100ms, a new preemption interval begins before the active one expires, and idletime use halts. This desired behavior protects foreground performance at high arrival rates, where even a moderate amount of idletime use can greatly decrease foreground performance.

Section 4.3.1 predicted that the lower bound for useful preemption intervals would be the service time of the resource. Such preemption intervals expire while their corresponding foreground request is still active, and consequently cannot control idletime processing. With preemption intervals shorter than the lower bound, the idletime scheduler degrades into a simple priority queue. The experiments with foreground UDP traffic in Chapter 6 have illustrated this property.

In another set of experiments that used TCP to transmit foreground traffic, the service time of the network interface was not a useful lower bound for preemption interval lengths. The following section will discuss this effect in detail.

7.1.3 Impact of the RTT on Foreground TCP

For UDP foreground traffic, the service time of the network stack acted as a lower bound for effective preemption lengths. Section 6.2.1 illustrates this behavior. Short preemption intervals of less than 0.05ms are sufficient in protecting UDP foreground traffic from the presence of TCP or UDP background transmissions.

However, foreground traffic using TCP for transmission required significantly longer preemption intervals to prevent reduced performance due to concurrent idletime traffic. Sections 6.2.2 and 6.3 presented evidence that effective idletime scheduling of TCP foreground traffic requires preemption interval lengths that exceed the RTT of the network link. With TCP foreground traffic, preemption intervals significantly shorter than TCP's RTT estimate – which has a resolution of 1.25ms on the benchmark platforms – are ineffective. The experiments over a simulated WAN link with 10ms delay in Section 6.3 illustrate this behavior. TCP foreground traffic in this scenario required preemption intervals longer than 20-30ms to prevent performance degradation.

The design of the idletime mechanism in Chapter 4 is independent of specific foreground arrival patterns, and its analysis did not predict the effect of TCP's RTT estimate on effective preemption interval ranges. A future investigation should validate these findings, and examine possible explanations for this TCP behavior. One hypothesis is that TCP's timers synchronize with the timers that control idletime transmissions, and therefore create a scenario where foreground TCP traffic incurs

frequent preemption costs, lowering performance. Another hypothesis is that timers in the *Dummynet* system, which simulated the WAN link in Section 6.3, may interfere with timers used to manage preemption intervals.

However, TCP's apparent dependence on longer preemption intervals does not invalidate the usefulness of the overall idletime mechanism. Preemption intervals longer than the RTT estimate successfully isolate foreground performance from the presence of idletime use. On the other hand, they do prevent the utilization of shorter, transient idle capacities, and will thus decrease idletime performance.

7.1.4 Congestion Control in the Background

Applying the idletime scheduler to networking, as described in Section 5.4, creates two distinct service environments. Foreground traffic continues to experience the traditional best effort Internet service model. Background traffic, however, is subject to preferential drops under congestion and experiences additional delays due to preemption intervals and low-priority queuing. It consequently experiences a more volatile overall network environment, where properties such as available bandwidth and propagation delays change in response to foreground load. These changes happen on timescales that are faster than for foreground traffic, and can affect the performance of transport protocols that transmit in the background.

When TCP transmits in the background, it can fail to reach performance that is comparable to that achieved at foreground priority. Queuing of idletime traffic during

preemption intervals can cause *ACK* compression [MOGUL1992], fluctuations in foreground traffic can rapidly change the available capacity for background use, and background delays through preemption intervals can inflate TCP's RTT estimate.

All these effects can decrease TCP's effectiveness. TCP operates by carefully monitoring the path characteristics, and slowly increasing its sending rate when it detects available capacity – or quickly dropping the sending rate in times of detected congestion. This mechanism is very effective when the path characteristics do not change on fast timescales. This is not true when TCP uses idletime network capacity for transmission, as previously described. Section 6.3.2 illustrates this limitation. During the experiments over a 100Mb/s link with 10ms delay, TCP background throughput is low, and can even become completely stalled at longer preemption intervals.

One approach to mitigating some of these effects relaxes the strict prioritization between foreground and background service [SHALUNOV2001]. Instead of completely stopping background traffic, this approach reserves a small fraction of the bandwidth for idletime use at all times. A similar approach that does not require reservations would simply transmit TCP packets that carry critical state information in the foreground. Both these approaches allow background TCP flows to maintain their *ACK* clocks during heavy foreground load, and can prevent frequent slow-starts that reduce performance. However, even with these improvements, TCP performance will suffer from rapid changes in available background bandwidth and RTT. Proposed

modifications to TCP, such as *TCP Santa Cruz* [PARSA1999], may alleviate some of these shortcomings, and could improve TCP performance in idletime capacity.

A transport protocol such as UDP, which is not congestion-controlled, does not suffer from such performance issues in the background. Because the idletime scheduler protects foreground traffic from the presence of high-rate background flows, using an aggressive transport protocol for background traffic is acceptable. Protocols that are more aggressive will react to changes in the network faster, and can thus increase background performance.

The experiments in Sections 6.2 and 6.3 illustrated that the idletime scheduler limits the impact of aggressive UDP background traffic, given sufficient preemption interval lengths. This allows maximizing background throughput without the possibility of decreasing foreground performance.

7.1.5 Effects of Speculative Optimizations

One unique characteristic of the disk subsystem is read-ahead optimization for sequential operations. Section 2.3.5 discussed why disabling such speculative optimization (including caching) during idletime use is required to limit the impact on foreground processing. However, disabling optimizations can lead to low idletime performance, even when the resource is only very lightly loaded with application-generated foreground requests.

The disk measurements in Section 6.1.2 also illustrated the opposite behavior. When a foreground process issues sequential reads, the disk subsystem aggressively issues additional, speculative read-aheads that the scheduler serves at foreground priority. This high frequency of read-aheads effectively stalls idletime use.

Additionally, speculative read-aheads generated for one foreground process may delay concurrent, application-issued disk operations of another foreground request. In some sense, read-aheads are less important that application-issued requests, because it is unknown whether their responses will benefit future processing.

Using idletime capacity for speculative read-aheads, as proposed in Section 2.3.5, addresses both these issues. They will execute concurrently with other idletime disk requests instead of stalling them, and cause application-issued requests to receive better service. A future version of the disk scheduler should investigate the performance effects of idletime read-aheads in detail.

7.2 Future Work

Although the experiments in Chapter 6 confirmed the general effectiveness of the present idletime scheduler, specific aspects, such as setting the preemption interval, or supporting advanced performance policies, can be improved. The current mechanism strives to minimize foreground performance decreases whenever possible. However, specific scenarios may tolerate a fixed decrease in foreground performance, and would

benefit from the corresponding increased idletime throughput. Section 7.2.1 discusses how a modification of the current state machine underlying the idletime scheduler can support such advanced policies.

A second direction for future research is mechanisms to choose effective preemption interval lengths automatically, depending on the present workload and user performance policy. Section 7.2.2 discusses possible approaches for such mechanisms, based on the beginnings of a framework already present in the current prototype implementation.

Finally, the idletime mechanism designed in Chapter 4 supports idletime use of spatially shared storage resources. The initial prototype implementation was limited to disk and network bandwidth, which are both temporally shared resources. Future extensions to the idletime implementation should include support for spatially shared resources, to validate the effectiveness of idletime scheduling with preemption intervals in this case. Idletime support for spatially shared resources will also enable research into speculative of idle resource capacity, as briefly outlined in Chapter 4. Section 4.2.4 discusses the required extensions to manage and merge speculative state kept maintained in storage capacity of spatially shared resources.

7.2.1 Idletime Scheduler Extensions

The implementation of the idletime scheduler described Chapter 5 bases its scheduling decisions strictly on its current state, such as its queue contents and preemption

interval timers. It does not accumulate usage history or track usage statistics, and past scheduling decisions have no influence on the present.

Consider the variant identified for implementation and evaluation in Chapter 5, with the state machine shown in Figure 7.3. It will always start a preemption interval by entering state P when leaving the F state, both when the resource becomes idle (event i) or when a background request is at the head of the queue (event b). Usage history – or any other external information – does not influence this scheduling decision.

This reliance on current state alone significantly simplifies operation and analysis of the idletime mechanism. However, it also eliminates possible optimizations of the idletime mechanism that could improve behavior. For example, in some scenarios, the user's foreground delay policy may allow for a specific, fixed decrease of aggregate foreground performance. Under such a policy, the idletime scheduler could skip the preemption interval in a controlled fashion when switching from foreground to idletime use. This can increase idletime performance.

Consider an example policy that permits a 10% reduction of foreground performance. Whenever the resource has served ten foreground requests without incurring a

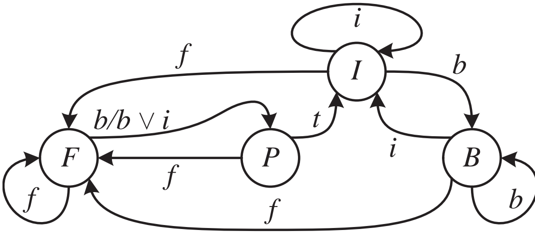


Figure 7.3. Variant of the idletime scheduler chosen for implementation.

preemption delay, it can immediately switch to idletime use. Even if it must immediately preempt the idletime request for a newly arriving foreground request, the aggregate foreground performance will still exceed 90%. It will have served eleven foreground requests incurring one preemption delay (of at most single service time). In general, the ratio between the number of serviced foreground requests that did not incur preemptions and the total number of preemptions bounds foreground performance.

Figure 7.4 shows the state diagram that would support such idletime scheduling. The thick, dotted arcs denote that this scheduler may either transition $F \rightarrow I$ directly on event *b* or *i*, or take the traditional transition $F \rightarrow P$, as the strict variant in Figure 7.3 does. The choice of when to skip the preemption interval (taking $F \rightarrow I$) and when to incur it (taking $F \rightarrow P$) supports different scheduler variants that may involve external state, such as event history, in their scheduling decision. The traditional state machine shown in Figure 7.3 cannot support such relaxed scheduling decisions.

The length of the event history acts as a moving averaging period. In the previous example, every ten foreground requests serviced without preemption cost count as a

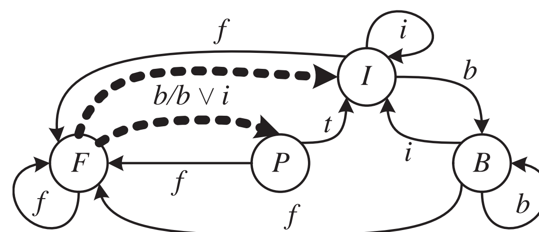


Figure 7.4. Idletime scheduler with relaxed transition into preemption intervals.

“credit” for skipping a preemption interval, while maintaining the required minimum performance. Suppose the event history records one hundred foreground requests served with five preemptions. The resource may still skip the next five preemption intervals while maintaining 90% aggregate foreground performance.

Long event histories potentially allow the relaxed scheduler to accumulate a large number of credits to skip preemption intervals. When the scheduler uses these credits in a short period, it may skip many preemption intervals, and cause transient foreground performance to decrease past the permitted reduction. For example, with five credits used back to back, the transient average foreground performance can dip as low as 50%, if all five accelerated idletime uses will incur preemptions. It may be useful to investigate *leaky bucket* schemes that further limit the number of saved skip credits, and the rate at which they may be spent.

The idea of accumulating and spending credits is similar to proportional-share schedulers discussed in Section 8.4 [WALDSPURGER1995]. Proportional-share schedulers allocate different fractions of resource capacity based on a weight distribution. In the context of the proposed idletime scheduler, such a mechanism would not manage resource use directly, but instead control the overheads of bypassing preemption intervals.

The relaxed observance of preemption intervals presented in this section can improve idletime performance, at a constant decrease in foreground performance. The details of

such idletime variants, including aggregation lengths, restrictions on spending skip credits, and the interaction between skipping preemption intervals and preemption interval lengths, are an area for future research.

7.2.2 Automatic Preemption Interval Tuning

The current idletime scheduler prototype requires manual specification of an appropriate preemption interval for a given resource and workload. One key improvement is a mechanism that automatically adapts the preemption interval based on observed resource and workload characteristics.

Effective idletime use requires amortization of preemption cost over a burst of foreground requests, which by definition incur at most a single preemption cost, and as a result bound idletime overhead. The idletime scheduler could measure burst and delay statistics, and thus automatically adjust the preemption interval length.

The prototype implementation described in Chapter 5 includes the beginnings of a framework to support such auto-tuning mechanisms. For each of the four possible states of the resource (I, F, B and P, see Figure 7.5), the scheduler maintains event

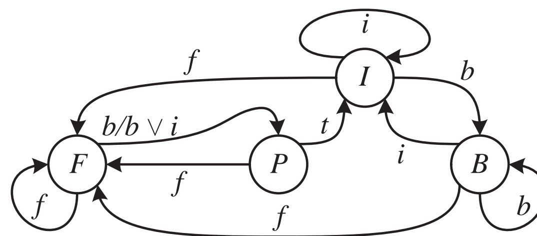


Figure 7.5. Variant of the idletime scheduler chosen for implementation.

counters for the *f*, *b*, *t* and *i* events. For example, whenever a new foreground request appears (event *f*) during a preemption interval (state *P*), the scheduler increases the *P[f]* counter by one.

A mechanism to adapt the preemption interval automatically can monitor these counters, to increase or decrease the interval length accordingly. For example, a rapidly increasing *B[f]* counter indicates that many idletime preemptions delayed foreground requests, and the preemption interval should be increased. Likewise, a steady increase in the *P[f]* and *I[f]* counters could allow a reduction of the preemption interval, to increase background throughput.

One interesting direction of future research is whether a TCP-like windowing mechanism can effectively manage preemption interval lengths based on these counters. For TCP, a segment loss serves as an indicator to decrease the congestion window. Similarly, an increase in *B[f]* can serve as an indicator for increasing the preemption interval length. In the absence of congestion losses, TCP slowly increases the window. The preemption interval could shrink slowly over time in the same way.

7.2.3 Spatially Shared Resources

Chapter 4 discussed the difference between temporally and spatially shared resources. Both prototype implementations in Chapter 5 focused on temporally shared resources.

Idletime use of spatially shared storage resources requires additional mechanisms, due to their inherent persistence. Traditionally, when a process obtains storage capacity, it is free to use it at any time thereafter. The kernel does not withdraw that storage capacity until the process explicitly returns it. This behavior remains unchanged for foreground use of capacity resources under idletime scheduling, but background use of idle capacity follows a different service model.

As described in Section 4.2, the operating system must reclaim unused storage allocated to idletime use when it is required to satisfy a newly arriving foreground request. This results in a service model where idletime storage can disappear at any time. Applications and services that wish to use idle capacity for storage must therefore gracefully adapt to these events.

Many existing applications may not execute under this service model for idletime storage. Furthermore, new operating system extensions must maintain the consistency of the overall system in the presence of preempted storage use. Finally, even with modified applications adapted to the service model of idletime storage, the operating system must provide further mechanisms to merge isolated idletime data into the regular, foreground state. Without this merge operation, idletime data could never become visible to regular processing, greatly reducing the usefulness of idletime storage.

Stateful resources, such as disk drives, are one challenge for idletime use of spatially shared capacity. File systems exploit locality by laying out data and metadata to reducing disk arm movement, increasing performance. A naïve implementation of an idletime mechanism could interfere with the layout of foreground data, and reduce performance. Section 2.3.6 discussed how idletime schedulers must prevent such side effects of background use. The presence of idletime use must not affect the layout policy of the foreground file system.

Another benefit of providing these additional mechanisms supporting transparent use of idle storage capacity is enabling *speculative* use of idletime capacity. The ability to store large amounts of data speculatively, without the possibility of interfering with regular, higher-priority storage requirements, allows straightforward support for aggressive optimizations such as caching and buffering. Combined with idletime use of temporally shared resources, such as the CPU or the network interface, these mechanisms provide an integrated framework for idletime processing. Successful speculations become visible to other processes in the system through the integrated, transparent merge operation supported by spatially shared resources.

7.2.4 Idletime Networking Improvements

Another area of improvements is the idletime network prototype. An earlier, experimental version of the idletime scheduler extended *ALTQ* to support different queuing strategies for the IP inbound queue [EGGERT2001A]. At the time, experimental evidence showed that inbound scheduling only offered minimal performance

improvements. Thus, the idletime inbound queuing code was not ported to the newer *ALTQ* release extended for idletime scheduling as described in Chapter 5.

However, the earlier idletime variant did not yet use preemption intervals. Preemption intervals during inbound network processing will delay delivery of background IP packets to higher layers, such as transport protocols and applications. Preemption intervals during inbound processing could therefore further reduce the impact on foreground traffic. Additional experiments are required to investigate the effects of idletime inbound network scheduling.

Another possible idletime networking improvement applies to idletime use of the TCP protocol. The current idletime mechanism starts a preemption interval whenever an outbound TCP segment enters the network driver. However, TCP is a bidirectional protocol based on a stream of receiver acknowledgements for pacing transmissions. Starting preemption intervals upon the *receipt* of such an acknowledgment – in addition to scheduling them when sending data segments – may further enhance foreground TCP performance under idletime scheduling.

7.3 Summary

This chapter discussed the overall results of the experimental evaluation. It compared the predicted performances based on the quantitative analysis from Chapter 4 against

the measured behavior from Chapter 6, and found that the simple performance predictor is often accurate to within 15% of the measured results.

The first part of this chapter discussed further cases where the predicted and measured scheduler behavior was similar. Examples include shielding foreground performances from the presence of idletime use, and the existence of upper and lower bounds for effective preemption interval ranges. It also analyzed cases where the predicted and measured behavior differed, such as interactions between TCP's congestion control mechanisms and the service model for idletime network transmissions.

A latter part of this chapter focused on directions for future research. One proposed improvement is a mechanism that automatically identifies appropriate preemption intervals for a given resource by observing its current workload. Another direction for future research is variants of the mechanism that can guarantee fixed reductions in foreground performance that may also increase idletime performance. Other possible extensions include prototype implementations of the scheduler for spatially shared resources.

8. Related Work

Related work falls into several broad categories. First, systems that prioritize resource use, such as hard and soft realtime systems. The second category comprises of idletime execution systems, including systems for process and data migration.. A third related area is mechanisms for maintaining state consistency, such as for distributed database systems. Finally, a fourth area is priority schemes proposed for specific resources or applications. The remainder of this section contrasts and compares these systems with the idletime scheduler.

8.1 Realtime Systems

In traditional computer systems, the correctness of the computation depends on producing the correct result for any given input. In a realtime system, correctness also depends on the result timing. Violating the timing constraints (deadlines) of a computation results in a critical failure, even if the logical result is accurate.

Systems that treat deadline violations as critical errors equivalent to a total system failure are commonly referred to as *hard* realtime systems. Another class of realtime systems supports *soft* realtime processing. They have relaxed timing constraints, and treat missed deadlines as undesirable, but not catastrophic. For both hard and soft realtime systems, the construction of an execution schedule that meets the deadlines of all tasks is critically important.

The spectrum of existing realtime systems – from hard realtime to soft “multimedia” realtime – is vast. The following sections will describe selected examples, and compare them to the idletime scheduler.

8.1.1 Examples

The *Spring* kernel [STANKOVIC1991] supports realtime execution on multiprocessor machines, and guarantees absolute predictability based on worst-case execution times. One processor of the system is dedicated to execution of the system kernel; the rest are available to execute user processes. One unique feature of *Spring* is its planning-based approach to resource scheduling. It eliminates blocking from the system, but depends on a detailed description of the resource use of all application programs. *Spring* offers predictable memory accesses by preloading and locking pages into physical memory, and by saving and restoring the translation look-aside buffer during context switches. (A similar technique for traditional systems is also effective [BALA1994]).

Nemesis [LESLIE1996] is a vertically structured operating system, where a microkernel implements only minimal task switching functionality. Shared libraries provide the bulk of in-kernel services of traditional operating systems at the application level. Thus, most processing on behalf of user processes is subject to scheduling by the microkernel. This allows accurate accounting of all computation. Traditional operating systems often cannot support precise accounting, and fail to charge internal processing to the process on which behalf it occurs.

Unlike *Spring*, *Nemesis* does not support hard realtime processing and does not require processes to specify their resource requirements in advance. Instead, *Nemesis* focuses on providing a consistent quality-of-service environment for multimedia (soft realtime) applications through a QoS manager. It notifies applications of changes to the service allocation, and expects them to adapt. Among other things, it signals to applications whether an increase in their resource share is due to a (temporary) increase in idle capacity, or to an actual change in the service allocation. Thus, Nemesis supports some notion of processing with idle capacity.

Eclipse/BSD [BRUNO1998][BRUNO1999] is similar to *Nemesis* in that it focuses on providing soft realtime service targeted at multimedia applications. Unlike the former, it requires explicit resource reservations through hierarchical CPU, disk, and network schedulers.

Realtime Mach [TOKUDA1990] is a microkernel-based operating system similar to *Eclipse/BSD*, but with support for hard realtime processes. Again, applications explicitly notify the system of their resource requirements through reservations.

AQUA [LAKSHMAN1998] is a kernel-level framework that allows cooperating processes to negotiate their CPU and network requirements with the kernel dynamically. If a resource becomes congested, *AQUA* notifies affected processes to allow service adaptation.

Omega [NAHRSTEDT1996] is an end system framework that supports soft realtime scheduling of CPU, memory and network I/O to provide end-to-end quality-of-service. *Omega* is similar to *AQUA*; processes dynamically negotiate their resource requirements with the system.

Scout [MOSBERGER1996] is a communication-oriented operating system based on the abstraction of data paths. *Scout* allocates threads to active paths according to a variety of schedulers, to vary the service model of the system. Idletime execution in *Scout* would require the addition of idletime paths combined with a thread scheduler supporting two service classes.

8.1.2 Discussion

All the realtime systems mentioned in the previous section differ in one or more of the following characteristics from a traditional, general-purpose operating system: predictability, resource requirement specifications, and admission control.

One difference is predictability, which requires time bounds on all resource operations and scheduling overheads. Without such bounds, guarantees for processing deadlines become impossible. Narrow bounds are desirable for higher system utilization. Defining time bounds on operations is difficult and usually hardware dependent – for example, the maximum time of a disk read operation depends on the disk drive model.

Predictability is not required for idletime use as defined in Chapter 4, although it might lower some preemption costs. With a known service time for a request, a scheduler may let an idletime request finish instead of preempting it when a regular request arrives. If the time-to finish of the idletime request is less than the preemption cost, this might decrease interference with regular use.

A second difference between regular and realtime systems is resource requirement specifications. Processes must disclose their future resource use to the system. In the basic case of a dedicated system, a programmer statically verifies that the system can satisfy all resource requirements of the various processes, and compiles a fixed schedule controlling resource use. Naturally, such a system will not support dynamic process creation, and is too limited for general-purpose use. More advanced realtime systems allow processes to disclose their planned resource use and deadlines at run-time. Such systems can automatically generate resource schedules based on these reservations. In both cases (explicit or automatic schedule generation), the workload of the system must be periodic. The resource requirements of dynamic workloads are difficult to predict, and their worst-case resource use may be unbounded.

The idletime scheduling mechanism proposed in this work does not mandate resource requirement specifications. If idletime tasks choose to specify their resource requirements, the system could optimize performance by not allocating available capacity to tasks that depend on fully loaded resources. However, this is an optional optimization of the idletime mechanism, and not a required feature.

When a new realtime process starts, the system dynamically verifies its execution feasibility, and rejects the process if execution would over-commit its resources. This admission control is the third characteristic of a realtime system. A general-purpose operating system does not need to perform admission control, because is neither offers resource reservations nor fixed deadlines. Dedicated realtime systems do not require admission control, because their workloads are static, with externally proven deadline guarantees.

With resource requirement specifications, more advanced systems can automatically generate schedules for periodic workloads. Such schedules require prioritized resource access. This aspect of realtime systems is very similar to the idletime scheduler, which also requires resource prioritization. Many of the prioritized schedulers proposed for realtime systems can implement this aspect of idletime processing. However, a key difference exists between realtime systems and the idletime mechanism proposed here. Realtime systems give preferential treatment to distinguished resource requests, to meet specified or implied service goals, whereas the idletime scheduler gives *less* service to distinguished idletime requests. In some sense, the current proposal is therefore the inverse of realtime service: the special class of idletime resource requests receives less-than-default service.

It is simple to convert a mechanism for realtime processing into one that establishes prioritization for idletime use (raise the default priority, use explicit notification to lower it). However, prioritization is not sufficient to establish full idletime use;

preemptability and isolation are also required, and realtime systems provide neither. For example, it is acceptable for a realtime system to continue processing a lower-priority request when a higher-priority one arrives, as long as it misses no deadlines. It may in fact be advantageous to avoid preemption, to increase resource utilization. Isolation is a concept that has no equivalent in a realtime system; side effects of execution at different priorities are always globally visible.

Another shortcoming of realtime systems is that that they require system-wide modifications. As discussed in Section 7.2.1, prioritization and preemptability alone require widespread changes across a processing hierarchy. On the other hand, the idletime scheduler with preemption intervals presented in this work is effective as a local change to key resource schedulers.

Furthermore, realtime systems focus on scheduling temporally shared resources. Idle-capacity use of spatially shared resources is a key part of this proposal that a realtime system does not address. Thus, although a realtime system can establish one requirement for idletime use (prioritization), two others are unsupported (preemptability and isolation). Furthermore, realtime execution requires predictability, resource requirement specification and admission control, all of which are unnecessary for idletime resource use.

8.2 Idletime Execution

All the previous techniques used idle local resources speculatively. Several other systems use idle remote resources for non-speculative purposes. One category of such systems is *process migration systems* (cycle harvesters), which push local processes to idle remote machines for faster execution. Another category is *data migration systems*, which push data to remote machines that execute a common process.

One major difference to this proposal is that these systems concentrate on detecting remote availability and then utilizing idle capacity for a single resource only, often the CPU. The proposed idletime scheduler, on the other hand, strives to utilize idle times of all resources independently of one another.

Another difference is that migration systems do not prioritize between idletime and regular processing. Instead, they treat idleness as a system-wide Boolean condition and thus only start idletime use when many system resources are completely idle. The idletime scheduler proposed here supports utilization of partially idle resources.

Although many systems (especially realtime systems, see Section 8.1) support high-priority resource access, few others offer the notion of idletime use. One of the few that does is a hierarchical CPU scheduler, where arbitrary threads can act as schedulers for other threads by donating part of their allocated CPU time [FORD1996]. One such scheduler explicitly supports background CPU use, similar to the POSIX idletime scheduler [POSIX1993].

8.2.1 Process Migration

Cycle harvesters schedule computations on a network of workstations, hoping to exploit idle remote resources to speed up local jobs. Historically, they have focused on utilizing remote CPU cycles (hence the name) and only utilized other remote resources indirectly. Cycle harvesting is especially effective for parallelizable jobs that can utilize multiple remote machines at once. However, even sequential jobs can benefit from remote idletime execution, where they do not have to compete for resources with other active processes.

The *V System* [THEIMER1985], *Condor* [LIZTKOW1988], *Benevolent Bandit* [FELDERMAN1989], the *Sprite System* [DOUGLIS1991], *DAWGS* [CLARK1992], and *Batrun* [TANDIARY1996] are cycle harvesters that support process re-migration, when a remote host under idletime use becomes unavailable. *Butler* [NICHOLS1987], a component of the *Andrew* system, is a transparent remote process execution facility that does not provide process migration, but simply terminates remote processes when a remote machine becomes unavailable.

Although cycle harvesters are similar in spirit to the proposed idletime scheduler – both approaches aim at reclaiming wasted capacity for useful work – several key differences exist. Cycle harvesters are often application-level or middleware solutions running on top of a conventional operating system without prioritized processing. Most of their shortcomings, such as migration overhead and idletime detection, are artifacts of that design.

Without prioritized resource use, cycle harvesters cannot effectively utilize machines with partially idle resources (bursty local workloads). Because migrated processes run at the same priority as regular ones on the remote machine, any migrated process can severely decrease regular performance on a remote machine. Thus, most harvesters only reclaim cycles from remote machines that are fully idle. The system presented in this proposal, however, supports prioritization and preemptability, and can utilize partial idle capacity.

Another consequence of the lack of prioritization is high migration costs. Whenever a remote machine becomes unavailable for idletime use, all remote processes on it must be re-migrated or terminated. Migration is a costly operation and decreases the performance of the remote machine during the migration period. Terminations are faster but still not instant, because the system must roll back to invalidate local state created by the terminated remote process. Additionally, partial work completed by terminated processes is lost.

With process migration systems, the finest-grained operation corresponds to the migration of a remote process. In the proposed system, on the other hand, an operation is a single resource request (e.g., sending a packet, reading a disk block). Consequently, the overhead of aborting idletime use in the proposed system is smaller, because the granularity of operations on the idle resource is finer-grained (e.g., waiting for disk read to finish vs. migrating an entire process).

High migration costs further reduce the chance for utilizing idle resources. For bursty remote workloads with short idle times, a cycle harvester could enter a state of thrashing, and spend all idle periods migrating process to and from a machine without making forward progress on the computation. Because the exact distribution of remote idle times is usually unknown, most cycle harvesters employ coarse heuristics and/or predictors [GOLDING1995][WYCKOFF1998] to find likely long idle periods. These techniques are effective in utilizing long, periodic idle periods (e.g., night hours), but fail to detect shorter, transient idle times due to quantization. They can hence fail to utilize some existing idle capacities of their target resource. The proposed system does not require such heuristics, because prioritization inherently establishes different service levels.

8.2.2 Data Migration

Unlike cycle harvesters, which push both code and data to an idle remote machine for execution, data migration systems only move data to idle peers for processing or storage. All remote machines participating in such a distributed system already run a copy of the same client process. Process migration systems offer more flexibility in remote idle-capacity processing, but data migration systems are simpler, can be platform-independent and have smaller preemption costs.

One popular subclass of such systems is application-level clients for distributed computation projects, such as protein folding and genome matching [LARSON2002], cryptographic code breaking, or searching for large prime numbers [HAYES1998] – or

even extraterrestrial life [KORPELA2001]. In these systems, all participants run the same client, and servers only migrate replicas of the data to be processed.

Other systems use unused remote memory as secondary storage, instead of a local disk [MINNICH1989][NARTEN1992][FEELEY1995][MARKATOS1996][KOUSSIH1999]. This can improve performance, because access times for remote memory over a local area network can be an order of magnitude lower than access times for local disk space.

As with process migration systems, the idletime mechanisms proposed in this paper can improve data migration systems by processing migrated data and communicating with remote peers during idle time.

8.2.3 Speculative Execution in Hardware

The proposed idea of using idle system resources productively is similar to some features found on modern microprocessors. A CPU with a superscalar architecture has multiple execution units, which allow it to execute multiple instructions per clock tick, increasing performance.

However, duplicating execution units cannot provide unlimited speedups, because superscalar execution requires a continuous instruction stream. Conditional branch instructions and indirect jumps limit such execution [TOUCH1991]. They introduce ambiguities into the instruction stream that require resolution before execution can proceed past them. Thus, execution units may remain idle until the CPU determines

whether to follow a specific branch. Instead, modern CPUs use speculative execution to process likely future instructions when the memory bus and some execution units are idle.

Speculative execution of instructions never decreases the execution speed of non-speculative processing, due to prioritized, preempted CPU resources (bus bandwidth, execution units). The CPU gives total priority to non-speculative instructions and immediately preempts any speculative processing for non-speculative execution. Hardware mechanisms eliminate preemption cost.

Furthermore, all side effects of a completed speculative instruction remain hidden until the processor can verify the prediction. For correctly predicted instructions, side effects become visible, whereas the CPU discards them for mispredictions. CPUs have hardware mechanisms to manage speculative state efficiently. Neither discarded nor committed operations delay regular processing.

Prioritized, preempted resource use, together with isolation of speculative side effects result in a worst-case performance that is identical to a CPU without speculation, even with constant mispredictions. For correct predictions, however, processor performance is improved.

Processor designs supporting *simultaneous multi-threading* (SMT) interleave execution of instructions of multiple threads, to increase processor utilization past

more traditional schemes that only exploit instruction-level parallelism. One speculative technique for SMT processors uses idle thread contexts to execute the less-likely branch of a predicted fork [WALLACE1998]. The authors report a 14-23% average speedup for single program performance on an SMT with eight thread contexts, for programs with a high rate of branch mispredictions. These results may indicate that sharing idletime capacity among multiple tasks may also increase performance.

The *Address Resolution Buffer (ARB)* [FRANKLIN1996] and the related decentralized *Speculative Versioning Cache* [GOPAL1998] allow reordering of memory-referencing instructions to exploit instruction-level parallelism. Traditional processors enforce a total order between memory references, whereas the *ARB* enforces total order only among references to the same address. The *ARB* also supports speculative loads/stores, dynamically unresolved loads/stores and memory renaming. The latter capabilities are similar to techniques required to support idle-capacity use of storage resources.

8.2.4 Speculative Execution in Software

Speculative execution has also been a part of some software systems, such as compilers or interpreters for programming languages. One example is a mechanism that speculatively interprets program branches in the *BaLinda* Lisp dialect, and assigns resources to speculative threads proportional to their relative likelihood [YEE1993].

A related compile-time technique speculatively executes some method calls of Java programs using idle multiprocessor capacity [CHEN1998]. For such methods, a speculative thread continues execution after the method's return point, using a predicted result value. The mechanism relies in part on properties of the Java virtual machine to shield threads from one another. The authors report significant speedups (up to a factor of 3) for data-parallel applications; only minor gains for control-flow-dependent programs.

Speculative execution has also been proposed in the area of information agents [BARISH2000] and decision flow optimization [HULL2000]. These approaches focus on generating good subtasks for speculative execution, but do not address the issue of executing them with idle capacities. This proposal, on the other hand, focuses on the operating system extensions required for non-interfering idletime use, but does not address generation of speculative subtasks. In that respect, the two mechanisms complement one another.

8.3 Isolation Techniques

Section 4.2.4 presented the principle of isolation, which requires that the side effects of idletime use must remain hidden until it finishes. The isolation principle virtualizes the operating system state. In an unmodified operating system, all processing operates on the same system state, transforming it over time. This can lead to incorrect

processing in the absence of isolation, if the side effects (state modifications) of idletime use become visible to regular processing.

One example of such a conflict is when an idletime process opens a network connection on a specific local port. If a regular process tries to perform the same operation later, the operating system must deny this request, because the port number is already in use.

This is what isolation prevents. Because all idletime processes execute on virtual operating system state (copy-on-write variant), the operating system state seen by regular processing remains unchanged, and the execution behavior remains unchanged from the basic case without idletime use present in the system.

If the system aborts an idletime process, it can discard the associated virtual state. However, the result of successful idletime use may become part of the regular operating system state. To prevent incorrect processing, this merge operation must be performed as an atomic operation with regard to other processing (regular and idletime).

Furthermore, conflicts between the regular and virtual operating system state can arise when regular processing modifies the same pieces of state as idletime processing. This is similar to processing of concurrent transactions in a database system, where the same data item may be involved in multiple transactions.

This section will first give a brief overview of related database mechanism, and then discuss specific approaches to managing state and evaluate their use for idletime isolation.

8.3.1 Database Concurrency Control

Transactions in database systems are atomic operations on the contents (state) of a database. Allowing multiple transactions to execute concurrently increases performance, but requires mechanisms that maintain database correctness.

Correctness depends on two conditions: integrity (defined through a set of constraints on the contents) and serializability. The latter requires database state changes to be equivalent to some serial execution of the given set of transactions.

A wide variety of mechanisms for concurrency control has been proposed [BERNSTEIN1981][KOHLER1981][THOMASIAN1998][BHARGAVA1999] . They generally fall into three groups: locking, timestamps and rollback.

One scheme to address concurrency control is locking all data items required for a transaction. When a data item carries a lock by another concurrent transaction, a transaction can wait, abort itself, or preempt the other transaction. This pessimistic scheme incurs the locking overhead even when transactions do not conflict. One issue with locking schemes is deadlock, a circular lock dependency among multiple

transactions. Various solutions, such as two-phase locking or ordered locks, can avoid deadlocks.

Another mechanism for concurrency control is timestamps on operations. Timestamps establish a fixed, serial processing order for all operations, guaranteeing consistency. Globally synchronized clocks are required. When conflicts arise, they are strictly resolved in timestamp-order. Some timestamp schemes use implicit locking to maintain consistency, whereas others use voting mechanisms that avoid centralized locks but incur increased communication overhead.

Concurrency control schemes using rollback differ from the two previous classes. No mechanism for conflict prevention is in effect during transaction processing. Instead, this scheme handles conflicts during commit time by rolling back all state changes, and then either aborting or restarting, when they detect a conflict. Rollback schemes are optimistic in that the basic assumption is that conflicts will be rare, and infrequent concurrency control during commit time is more efficient than employing an a priori scheme on every transaction.

8.3.2 Discussion

The proposed mechanism for idletime isolation could benefit from database concurrency-control techniques. Operating system processes are similar to database transactions. By mapping operating system processes directly into transactions on a database, relevant techniques immediately apply. Even when this is not possible,

concurrency-control techniques can improve the critical operation maintaining the isolation principle – merging of virtual state.

8.3.2.1 Processes as Transactions

At some level, process execution in an operating system and transaction processing in a database system are similar. Both allow multiple, concurrent entities (processes and transactions) to perform operations on shared state. However, concurrency control mechanisms for database systems may not directly apply to operating systems, due to a few key differences.

State conflicts in operating system processing are relatively rare. First, because processes usually spend a good part of their time in user-space processing private, usually unshared state. Second, multiprocessors were rare, and systems had consequently only one active physical thread of execution, even though simulating multiple threads of control through CPU scheduling. Thus, the operating system could lock state through blocking interrupts, a fast operation. Because interrupt blocking is limited to single CPUs, with multi-processors becoming more common, operating systems must support finer-grained locking [SCHIMMEL1994][LEHEY2001]. Scheduling benefits, which allow more than one CPU to execute kernel code concurrently, can compensate for the additional locking overhead compared to blocking interrupts.

Another issue is that concurrency-control mechanisms in databases must be general enough for a wide variety of dynamic application domains. On the other hand, the uses for concurrency-control mechanisms in an operating system are well-known and static, so simplified special-case mechanisms are worth deploying, e.g., for the process lists, device queues, etc.

Furthermore, in the idletime model, idletime use has a lower priority than regular processing and is preemptable. Although some concurrency control mechanisms support similar prioritized models, e.g., for realtime databases [HARITSA1992][YU1994][LINDSTROM2000], they are not immediately applicable to prioritize idletime use.

In databases, one correctness criterion is the existence of a serialized execution of the same transactions. The valid execution order of a set of operating system operations in the presence of speculations is much more constrained. The order of regular operations on operating system state must be unchanged from the basic case when speculations are present, and the intermediary operating system states must be identical, as described in Section 4.2.4. Database mechanisms enforcing conventional serializability may not satisfy these stricter requirements.

8.3.2.2 Concurrency Control for State Merging

Although processing in general databases and operating systems is very similar at a high level, the operations required to support idletime isolation have unique properties.

Many mechanisms for database systems either do not support these properties, or only support them inefficiently, incurring additional overheads due to their generality. However, specialized modifications of some of the basic concurrency-control techniques, such as lock-based schemes, can support isolation. This section will discuss this in more detail, and relate the proposed isolation mechanism from Section 4.2.4 to its database equivalents.

The atomic merge operation after a successful speculation is one instance where concurrency-control mechanisms from databases may apply. This state merge is a strictly confined operation. First, only two sets of data are involved, regular and idletime. It is rare that two idletime processes finish at the same time, and they are independent and can be committed in any order. Second, regular state has priority over idletime state. If a piece of regular state has changed during idletime processing, the merge cannot complete, and the system must discard the result of the idletime operation. Third, even if an idletime process runs to completion, merging its state may not have benefit if continued regular processing has updated the same state already.

These properties make optimistic, rollback-based ideas unsuitable for the state merge operation. Such mechanisms would merge idletime state before it finishes, because they assume the absence of conflicts and success on termination. Conflicts trigger rollbacks, which cause regular processing delays.

Timestamp-based mechanisms are also not well suited to this scenario. Timestamps provide a serial execution order for transactions. However, timestamps do not capture the constraints of the state merge. Regular state always overrides idletime state.

Lock-based mechanisms, on the other hand, are very applicable. A single lock for the whole state is the simplest solution. In effect, this preempts regular use for the duration of the merge operation, and will therefore decrease regular performance. However, only finishing idletime processes with successful results incur this overhead. Such processes potentially improve regular performance, for example, when a speculative task succeeded. Thus, the speculation gain may compensate for the locking overhead.

More advanced schemes use multiple locks for different parts of the operating system state, and further minimize the locking cost. For example, if an idletime task will only changed the “network” part of the operating system state, it would only need to acquire the “network” lock. Regular processing that does not involve the “network” state can continue execution during the merge. This is similar to the copy-on-write approach for idletime state management, where multiple locks for different pieces of state allow merging idletime revisions.

In the extreme case, each data item in the operating system state would have a separate lock. Clearly, this is infeasible due to the space overhead. An adequate mechanism will

probably utilize multiple, fine-grained locks for logically separate parts of the state space, optimizing common processing patterns.

8.4 Priority Schemes

Another area of related work focuses on prioritized service for specific resources or services, such as network traffic or disk I/O. Many proposals for prioritized service focus on realtime systems (see Section 8.1). This section will concentrate on other, non-realtime proposals and compare them to the idletime scheduler with preemption intervals.

A previous paper has investigated the idea of providing idletime network service at the application layer by distinguishing between regular and background web transactions [EGGERT1999]. This idea came out of the *LSAM* project [TOUCH1998], which used this capability for speculative background multicasting of web transactions to pre-load self-organizing, distributed caches with popular content. [ALMEIDA1998] also proposed both application-level and in-kernel mechanisms for web traffic prioritization. A kernel-level mechanism for idletime networking was the result of an earlier instantiation of the idletime processing model [EGGERT2001A][EGGERT2001B]. This earlier work did not use a preemption interval to provide non-interference, and was therefore less effective than the approach presented here. It also failed to utilize significant amounts of idle capacity for background use. Chapter 3 evaluated both preliminary systems in detail.

Several other proposals establish prioritized network service, as discussed in Section 5.4.1. The idea of marking packets according to their priority is present in the original Internet architecture [POSTEL1981], as well as several link-layer technologies, such as the ATM *cell loss priority* bit [ATM1999] or the Frame Relay *discard eligible* bit [THIBODEAU1998]. Proposed Internet extensions for differentiated service [BLAKE1998] give different per-hop forwarding service to packets of different priority classes [CLARK1998][DAVIE2002].

Other proposals address traffic prioritization at the transport layer. One proposed system to preload web caches uses a simulated, connectionless datagram protocol (essentially UDP) together with low-priority forwarding [DAVISON2000].

TCP Nice [VENKATARAMANI2002] is a sender-side modification of the traditional TCP congestion control algorithm that establishes low-priority service. It supports background replication of web content in the *NPS* system [KOKKU2003]. *NPS* is similar in scope and design to the earlier *LSAM* system [TOUCH1998], and *TCP Nice* corresponds to a transport-layer equivalent of the application-layer background traffic mechanism in [EGGERT1999]. *TCP-LP* [KUZMANOVIC2003] is another TCP modification similar to *TCP Nice*.

MulTCP allows users to assign weights (priorities) to different connections [CROWCROFT1998]. Weights correspond to proportional shares of the available bandwidth along a congested path.

The *stride scheduler* is a proportional-share mechanism [WALDSPURGER1995]. Experiments show that it successfully allocates different shares of the managed resource to different users, such as for network transmissions in the Linux kernel. The key difference between proportional-share approaches – such as this scheduler and *MulTCP* – compared to the idletime scheduler with preemption intervals is starvation preemption. True idletime use requires zero-length shares, which are often unsupported. [SULLIVAN2000] proposes an extension of the stride scheduler that further minimizes the impact of concurrent use on the relative performances observed by users.

Migrating Sockets follow a different approach to establish different network service levels [YAU1998]. They push most protocol processing out of the kernel and into user-level processes. A rate-controlling network scheduler then controls transmissions to meet pre-defined quality-of-services parameters. In a sense, this design is the inverse to the proposed idletime scheduler that minimizes changes to the network stack, and targets hidden scheduling (processing due to asynchronous events) at the lowest possible layer. *Migrating Sockets*, on the other hand, completely re-designs the network stack, and minimizes hidden scheduling by doing protocol processing at the user level, where CPU scheduling controls it.

Various application-level mechanisms strive to provide support for a background service class. *MS Manners* [DOUCEUR1999] is an application-level service that monitors the progress of cooperating background applications. It is based on the

principle that an observed drop in performance must be due to resource contention. Consequently, when *MS Manners* detects a drop in background performance, it notifies the cooperating background applications to reduce their aggressiveness further. Though reported to be effective, the mechanism does not prevent reductions in foreground performance, and instead reacts to their presence. It is furthermore requires application modifications to monitor and report progress.

Two other application-level mechanisms aim at providing background network transmissions. The *Mozilla* web browser continually monitors its own network transmissions [FISHER2002]. When *Mozilla* does not transmit user-requested data, it will prefetch likely future web pages in the background. One major limitation of this mechanism is its inability to detect network transmissions by other applications. Background page prefetches can thus delay other network transmissions.

Microsoft's *Background Intelligent Transfer Service* (BITS) [MICROSOFT2002] is a network transmission scheduler that supports background web transmissions using HTTP. Applications submit transmission requests, and receive notifications on request completion. *BITS* does not support complete starvation, i.e., background transmissions will never completely halt even when foreground transmissions require all available bandwidth, and thus fails to isolate foreground performance. Furthermore, it is a reactive mechanism similar to *MS Manners*, and requires several seconds to adjust the rate of background traffic, further increasing interference with concurrent foreground work.

Middleware approaches for quality-of-service support, interposed between the application and the kernel, promise effective support for different service profiles without kernel changes, and only minor changes to the applications [ABDELZAHER1999]. However, by processing transmissions outside the kernel, scheduling traffic on a per-packet scale becomes unattainable. Instead, middleware approaches control traffic at a coarser granularity (e.g., flow-based), similarly to application-level approaches.

Another issue with middleware or application-level proposals is performance at Gigabit speeds. Current workstations based on the common PC architecture (or similar) already fail to drive Gigabit link layers at line rate. Adding additional context switches between kernel and user-level processing can further decrease performance. On the other hand, the idletime scheduler presented here is an in-kernel mechanism that does not incur additional context switches. The experimental evaluation in Section 6.2 verifies that the mechanism can support Gigabit speed without significant overhead.

9. Conclusion

Common workloads on many computer systems rarely utilize resources fully. Even when the bottleneck resource becomes fully loaded, other resources often have idle capacity available. Using this idle capacity for productive work – without interfering with the ongoing foreground work – can improve overall system efficiency and user-perceived performance.

Many current operating systems do not support such idletime use of available capacity. Their schedulers strive to guarantee fairness and prevent starvation, but do not support prioritized use, preemption, or full isolation of concurrent tasks from one another. Furthermore, many systems implicitly prioritize internal processing, such as interrupt handling. All these factors cause delays for regular foreground use that render idletime service based on traditional operating system mechanisms ineffective. Chapter 2 discussed the challenges associated with idletime use in detail, and Chapter 3 presented preliminary work in application- and kernel-level mechanisms for idletime network scheduling. Although effective in establishing different levels of service, the preliminary mechanisms did not successfully isolate foreground traffic from the presence of background transmissions. They also failed to utilize significant available capacity for background use, and do not generalize from network scheduling to arbitrary resources.

The key contribution of this work is a general, resource-independent scheduler that can support idletime use while minimizing delays of the foreground workload. This scheduler addresses the limitations of the preliminary mechanisms in Chapter 3. It selectively relaxes the work conservation property for idletime tasks during a period called the preemption interval. The length of a preemption interval is a variable that allows tuning of the mechanism to accommodate specific resource characteristics, workloads, and user policies. Chapter 4 formally defined a model for resource processing, and described the required properties of an effective idletime scheduler in terms of the model. It also analyzed the proposed scheduler quantitatively, and presented a simple model that can predict the expected regular and idletime performances for given resources and workloads.

One key advantage of the idletime scheduler is deployability as a localized modification to selected schedulers. Other proposals supporting prioritized resource use require widespread changes to many parts of the operating systems, and often depend on application modifications or cooperation. Chapter 5 discussed several variants of idletime scheduling that satisfy the principles defined in Chapter 4, and identified one variant for implementation and evaluation. Later sections of Chapter 5 described the prototype implementation of idletime scheduling for the disk and network schedulers of the FreeBSD operating system. For each resource, changes to a single scheduler established idletime use.

Chapter 6 presented a detailed experimental investigation of the prototype disk and network scheduler implementations for a variety of different workloads. All scenarios modeled the worst case of an unlimited idletime workload, and measured regular and idletime throughput and latencies. In most of these worst-case scenarios, the idletime scheduler was effective in shielding regular processing from concurrent idletime use, incurring a maximum of 10-15% performance impact.

Chapter 6 discussed how the experimentally observed behavior of the idletime scheduler conformed to the theoretical model. It includes a detailed comparison between the measured performances of selected scenarios and performance predictions based on the quantitative analysis in Chapter 4. Despite its simplicity, the quantitative model generally predicted measured performances to within 5-15%.

The experiments also verified that the length of the preemption interval is an effective control mechanism that allows trading a reduction in idletime performance for an increase in foreground performance. It enables adaptation of the scheduling behavior to resource, workload, and user policy characteristics. Longer preemption intervals increase foreground performance through a corresponding reduction in idletime processing performance. Chapter 7 also discusses future scheduler extensions. Examples include mechanisms that automatically adapt the idletime scheduler to conform to user delay policies, based on the currently observed workloads.

Many other systems strive to utilize idle resource capacity, such as process migration systems, caches, or prefetchers. Other proposals, such as realtime systems, also offer prioritized resource service. However, the majority of realtime systems depend on explicit, deadline-based resource reservations, which cannot easily describe interactive, aperiodic tasks that are common on interactively used computer systems. Chapter 8 discussed these and other related proposals, and compared them to the idletime scheduler presented in this work.

Bibliography

[ABDELZAHER1999] Tarek F. Abdelzaher and Kang G. Shin. QoS Provisioning with qContracts in Web and Multimedia Servers. Proc. IEEE Real-Time Systems Symposium, Phoenix, AZ, USA, December 1-3, 1999, pp. 44-53.

[ACHARYA1999] Anurag Acharya and Sanjeev Setia. Availability and Utility of Idle Memory in Workstation Clusters. Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Atlanta, GA, USA, May 1-4, 1999, pp. 35-46.

[AKYÜREK1995] Sedat Akyürek and Kenneth Salem. Adaptive Block Rearrangement. ACM Transactions on Computer Systems, Vol. 13, No. 2, May 1995, pp. 89-121.

[ALMEIDA1998] Jussara Almeida, Mihaela Dabu, Anand Manikutty and Pei Cao. Providing Differentiated Levels of Service in Web Content Hosting. Proc. SIGMETRICS Workshop on Internet Server Performance, Madison, WI, USA, June 23, 1998, pp. 91-102.

[AMDAHL1967] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. Proc. AFIPS Joint Computer Conference, Atlantic City, NJ, USA, April 18-20, 1967, pages 483-485.

[APACHE1995] Apache Software Foundation. Apache HTTP Server Project. http://www.apache.org/

[ARON1998] Mohit Aron and Peter Druschel. TCP: Improving Startup Dynamics by Adaptive Timers and Congestion Control. Technical Report TR98-318, Department of Computer Science, Rice University, Houston, TX, USA, June 3, 1998.

[ARON2000] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. ACM Transactions on Computer Systems (TOCS), Vol. 18, No. 3, August 2000, pp. 197-228.

[ATM1999] ATM Forum. ATM Forum Traffic Management Specification Version 4.1. AF-TM-0121.000, March 1999.

[BALA1994] Kavita Bala, M. Frans Kaashoek and William E. Weihl. Software Pre-fetching for Translation Lookaside Buffers. Proc. USENIX Symposium on Operating System Design and Implementation (OSDI), Monterey, CA, USA, November 14-17, 1994, pp 243-254.

[BARISH2000] Greg Barish, Craig A. Knoblock and Steven Minton. Speculative Execution for Information Agents. Proc. National Conference on Artificial Intelligence (AAAI), Austin, TX, USA, July 30 - August 3, 2000, pp. 1062.

[BERNSTEIN1981] Philip A. Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. ACM Computing Surveys, Vol. 13, No. 2, June 1981, pp. 185-221.

[BHARGAVA1999] Bharat Bhargava. Concurrency Control in Database Systems. IEEE Transactions on Knowledge and Data Engineering, Vol. 11, No. 1, January/February 1999, pp. 3-16.

[BLAKE1998] Steven Blake, David Black, Mark Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An Architecture for Differentiated Services. RFC 2475, December 1998.

[BORMANN1999] Carsten Bormann. Providing Integrated Services over Low-bitrate Links. RFC 2689, September 1999.

[BRAKMO1995] Lawrence S. Brakmo and Larry L. Peterson. Performance Problems in BSD4.4 TCP. ACM SIGCOMM Computer Communication Review , Vol. 25, No. 5, October 1995, pp. 69-86.

[BRESLAU2000] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in Network Simulation. IEEE Computer, Vol. 33, No. 5, May 2000, pp. 59-67.

[BRUNO1998] John Bruno, Eran Gabber, Banu Özden, and Abraham Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. Proc. USENIX Annual Technical Conference, New Orleans, LA, USA, June 15-19, 1998, pp. 235-246.

[BRUNO1999] John Bruno, José Brustoloni, Eran Gabber, Banu Özden, and Abraham Silberschatz. Retrofitting Quality of Service into a Time-Sharing Operating System. Proc. USENIX Annual Technical Conference, Monterey, CA, USA, June 6-11, 1999, pp. 15-26.

[CARLBERG2001] Ken Carlberg, Panos Gevros and Jon Crowcroft. Lower than Best Effort: a Design and Implementation. Proc. ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean, San Jose, Costa Rica, April 3-5, 2001, pp. 244-265.

[CHANG1999] Fay Chang and Garth A. Gibson. Automatic I/O Hint Generation through Speculative Execution. Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, LA, USA, February 22-25, 1999, pp. 1-14.

[CHEN1998] Mike Chen and Kunle Olukotun. Exploiting Method-Level Parallelism in Single-Threaded Java Programs. Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT), Paris, France, October 12-29, 1998, pp. 176-184.

[CHO1998] Kenjiro Cho. A Framework for Alternate Queuing: Towards Traffic Management by PC-UNIX Based Routers. Proc. USENIX Annual Technical Conference, New Orleans, LA, USA, June 15-19, 1998, pp. 247-258.

[CLARK1988] David Clark. The Design Philosophy of the DARPA Internet Protocols. ACM SIGCOMM Computer Communication Review, Vol. 18, No. 4, 1988, pp. 106-114.

[CLARK1992] Henry Clark and Bruce McMillin. DAWGS – A Distributed Compute Server Utilizing Idle Workstations. Journal of Parallel and Distributed Computing, Vol. 14, 1992, pp 175-186.

[CLARK1998] David Clark and Wenjia Fang. Explicit Allocation of Best-Effort Packet Delivery Service. IEEE/ACM Transactions on Networking, Vol. 6, August 1998, pp. 362-373.

[COHEN2000] Edith Cohen and Haim Kaplan. Prefetching the Means for Document Transfer: A New Approach for Reducing Web Latency. Proc. IEEE INFOCOM, Tel Aviv, Israel, March 26-30, 2000, pp. 854-863.

[CROWCROFT1998] Jon Crowcroft and Philippe Oechslin. Differentiated End-to-End Internet Services using a Weighted Proportional Fair Sharing TCP. ACM SIGCOMM Computer Communication, Vol. 28, No. 3, July 1998, pp. 53-67.

[DAVISON2000] Brian D. Davison and Vincenzo Liberatore. Pushing Politely: Improving Web Responsiveness One Packet at a Time. Performance Evaluation Review, Vol. 28, No. 2, September 2000, pages 43-49.

[DIKE2001] Jeff Dike. User Mode Linux. Proc. Linux Symposium, Ottawa Canada, July 25-28, 2001.

[DOUCEUR1999] John R. Douceur and William J. Bolosky. Progress-based regulation of low-importance processes. Proc. ACM Symposium on Operating System Principles (SOSP), Kiawah Island Resort, SC, USA, December 12-15, 1999, pp. 247-260.

[DOUGAN1999] Cort Dougan, Paul Mackerras and Victor Yodaiken. Optimizing the Idle Task and Other MMU Tricks. Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, LA, USA, February 22-25, 1999, pp. 229-237.

[DOUGLIS1991] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. Software – Practice and Experience (SPE), Vol. 21, No. 8, 1991, pp. 757-785.

[DRUSCHEL1996] Peter Druschel and Gaurav Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. Proc. USENIX Symposium on Operating System Design and Implementation (OSDI), Seattle, WA, USA, October 28-31, 1996, pp. 261-275.

[EGGERT1999] Lars Eggert and John Heidemann. Application-Level Differentiated Services for Web Servers. World Wide Web Journal, Vol. 3, No. 2, 1999, pp. 133-142.

[EGGERT2001A] Lars Eggert and Joseph D. Touch. End-System Support for Idletime Networking. ISI Technical Report ISI-TR-559, USC Information Sciences Institute, May 2001.

[EGGERT2001B] Lars Eggert. Speculative Use of Idle Resources. Ph.D. Dissertation Proposal, ISI Technical Report ISI-TR-560, USC Information Sciences Institute, October 2001.

[FEELEY1995] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy and Chandroman A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. Proc. ACM Symposium on Operating System Principles (SOSP), Copper Mountain Resort, CO, USA, December 3-6, 1995, pp. 201-212.

[FELDERMAN1989] Robert E. Felderman, Eve M. Schooler and Leonard Kleinrock. The Benevolent Bandit Laboratory: A Testbed for Distributed Algorithms. IEEE Journal on Selected Areas in Communications (JSAC), Vol. 7, No. 2, February 1989, pp. 303-311.

[FISHER2002] Darin Fisher. Mozilla Link Prefetching FAQ. October 14, 2002.

[FITZGERALD1986] Robert Fitzgerald and Richard F. Rashid. The Integration of Virtual Memory Management and Interprocess Communication in Accent. ACM Transactions on Computer Systems, Vol. 4, No. 2, May 1986, pp. 147-177.

[FORD1996] Bryan Ford and Sai Susarla. CPU Inheritance Scheduling. Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA, USA, October 28-31, 1996, pp. 91-105.

[FRANKLIN1996] Manoj Franklin and Gurindar S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. IEEE Transactions on Computers, Vol. 45, No. 5, May 1996, pp. 552-571.

[GOLDING1995] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. Proc. USENIX Technical Conference, New Orleans, LA, USA, January 16-20, 1995, pp. 201-212.

[GOPAL1998] Sridhar Gopal, T.N. Vijaykumar, James E. Smith and Gurindar S. Sohi. Speculative Versioning Cache. Proc. Symposium on High-Performance Computer Architecture, Las Vegas, NV, USA January 31 - February 4, 1998, pp. 195-205.

[GUPTA1997] Alok Gupta, Dale O. Stahl and Andrew B. Whinston. Priority Pricing of Integrated Services Networks. Internet Economics, L. W. McKnight and J. P. Bailey (editors), MIT Press, 1997, pp. 323-352.

[HARDIN1968] Garrett Hardin. The Tragedy of the Commons. Science, Vol. 162, 1968, pp. 1243-1248.

[HARITSA1992] Jayant R. Haritsa, Michael J. Carey and Miron Livny. Data Access Scheduling in Firm Realtime Databases. Realtime Systems, Vol. 4, No. 3, 1992, pp. 203-241.

[HARKINS1998] Dan Harkins and D. Carrell. The Internet Key Exchange (IKE). RFC 2409, November 1998.

[HAYES1998] Brian Hayes. Collective Wisdom. American Scientist, Vol. 86. No. 2, March-April 1998, pp. 118-122.

[HP1995] Netperf: A Network Performance Benchmark. Revision 2.0. Technical Report, Information Networks Division, Hewlett-Packard Company, February 15, 1995.

[HUGHES2001] Amy S. Hughes and Joseph D. Touch. Expanding the Scope of Prefetching through Inter-Application Cooperation. Proc. International Web Content Caching and Distribution Workshop (WCW), Boston, MA, USA, June 20-22, 2001, pp 129-130.

[HUGHES2002] Amy S. Hughes. Enhancing Network Object Caches through Cross-Domain Cooperation. Ph.D. Dissertation, Department of Computer Science, University of Southern California, December 2002.

[HULL2000] Richard Hull, Francois Llirbat, Bharat Kumar, Ganz Zhou, Gouzhu Dong and Jianwen Su. Optimization Techniques for Data-Intensive Decision Flows. Proc. International Conference on Data Engineering (ICDE), San Diego, CA, USA, February 26 - March 3, 2000, pp 281-292.

[HUWIG2003] Kurt Huwig. Divide and Conquer – Virtual Server Contexts in Practical Applications. Linux Magazine, February 2003, pp. 32-35.

[IYER2001] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. Proc. ACM Symposium on Operating Systems Principles (SOSP), October 21-24, 2001, Chateau Lake Louise, Banff, Alberta, Canada, pp. 117-130.

[JACOBSON1988] Van Jacobson. Congestion Avoidance and Control. Proc. ACM SIGCOMM, August 16-19, 1988, Stanford, CA, USA, pp. 314-329.

[JACOBSON1992] Van Jacobson, Robert Braden and Dave Borman. TCP Extensions for High Performance. RFC 1323, May 1992.

[DAVIE2002] B. Davie, A. Charny, J. C. R. Bennett, K. Benson, J. Y. Le Boudec, W. Courtney, S. Davari, V. Firoiu and D. Stiliadis. An Expedited Forwarding PHB (Per-Hop Behavior). RFC 3246, March 2002.

[JINMEI1998] Tatuya Jinmei, Kazu Yamamoto, Jun-ichiro Hagino, Munechika Sumi-kawa, Yoshinou Inoue, Kazushi Sugyo and Soichi Sakane. An Overview of the KAME Network Software: Design and Implementation of the Advanced Internetworking Platform. Proc. Annual Conference of the Internet Society (INET), San Jose, CA, USA, June 22-25, 1998.

[KAMP2000] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the Omnipotent Root. Proc. System Administration and Networking Conference (SANE), Maastricht, The Netherlands, May 22-25, 2000.

[KING1990] Richard P. King. Disk Arm Movement in Anticipation of Future Requests. ACM Transactions on Computer Systems, Vol. 8, No. 3, 1990, pp. 214-229.

[KOHLER1981] Walter H. Kohler. A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems. ACM Computing Surveys, Vol. 13, No.2, June 1981, pp. 149-183.

[KOKKU2003] Ravi Kokku, Praveen Yalagandula, Arun Venkataramani and Mike Dahlin. NPS: A Non-interfering Deployable Web Prefetching System. Proc. USENIX Symposium on Internet Technologies and Systems (USITS), March 26-28, 2003, Seattle, WA, USA.

[KORPELA2001] Eric Korpela, Dan Werthimer, David Anderson, Jeff Cobb and Matt Lebofsky. SETI@home: Massively Distributed Computing for SETI. IEEE Computing in Science and Engineering, Vol. 3, No. 1, January/February 2001, pp. 78-83.

[KOUSSIH1999] Samir Koussih, Anurag Acharya and Sanjeev Setia. Dodo: A User-level System for Exploiting Idle Memory in Workstation Clusters. Proc. IEEE International Symposium on High Performance Distributed Computing (HPDC), Redondo Beach, CA, USA, August 1999, pp. 301-308.

[KUZMANOVIC2003] Aleksandar Kuzmanovic and Edward W. Knightly. TCP-LP: A Distributed Algorithm for Low Priority Data Transfer. Proc. IEEE INFOCOM, San Francisco, CA, USA, April 2003, pp. 1691-1701.

[KWAK1999] Hantak Kwak, Ben Lee, Ali R. Hurson, Suk-Han Yoon and Woo-Jong Hahn. Effects of Multithreading on Cache Performance. IEEE Transactions on Computers, Vol. 48, No. 2, February 1999, pp. 176-184.

[LAI1996] Kevin Lai and Mary Baker. A Performance Comparison of UNIX Operating Systems on the Pentium. Proc. USENIX Annual Technical Conference, San Diego, CA, USA, January 22-26, 1996, pp. 265-278.

[LAKSHMAN1998] K. Lakshman, Raj Yavatkar and Raphael Finkel. Integrated CPU and Network-I/O QoS Management In An Endsystem. Computer Communications, Vol. 21, No. 4, April 1998, pp. 325-333.

[LAMPSON1980] Butler Lampson and David Redell. Experience with Processes and Monitors in Mesa. Communications of the ACM, Vol. 23, No. 2, February 1980, pp. 105-117.

[LARSON2002] Stefan M. Larson, Christopher D. Snow, Michael Shirts, and Vijay S. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. Computational Genomics, Horizon Press, 2002.

[LEHEY2001] Greg Lehey. Improving the FreeBSD SMP Implementation. Proc. FREENIX Track: USENIX Annual Technical Conference, Boston, MA, USA, June 25-30, 2001.

[LESLIE1996] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Bar-ham, David Evers, Robin Fairbairns and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. IEEE Journal on Selected Areas In Communications (JSAC), Vol. 14, No. 7, September 1996, pp. 1280-1297.

[LEVER2000] Chuck Lever, Marius Aamodt Eriksen and Stephen P. Molloy. An analysis of the TUX web server. CITI Technical Report 00-8, Center for Information Technology Integration, University of Michigan, November 16, 2000.

[LINDSTROM2000] Jan Lindstrom and Kimmo Raatikainen. Using Importance of Transactions and Optimistic Concurrency Control in Firm Realtime Databases. Proc. International Conference on Realtime Systems and Applications (RTCSA), Cheju Island, South Korea, December 12-14, 2000.

[LIZTKOW1988] Michael J. Liztkow, Miron Livny and Matt W. Mutka. Condor – A Hunter of Idle Workstations. Proc. International Conference on Distributed Computing Systems (ICDCS), San Jose, CA, USA, June 13-17, 1988, pp. 104-111.

[MARKATOS1996] Evangelos P. Markatos and George Dramitinos. Implementation of a Reliable Remote Memory Pager. Proc. USENIX Annual Technical Conference, San Diego, CA, USA, January 22-26, 1996, pp. 177-190.

[MATTHEWS1997] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang and Thomas E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. Proc. ACM Symposium on Operating Systems Principles (SOSP), Saint Malo, France, October 5-8, 1997, pp. 238-251.

[MAY1997] Martin May, Jean-Chrysostome Bolot, Christophe Diot and Alain Jean-Marie. 1-Bit Schemes for Service Discrimination in the Internet: Analysis and Evaluation. Technical Report RR-3238, INRIA, Sophia Antipolis, France, August 1997.

[MCKUSICK1984] Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. ACM Transactions on Computer Systems, Vol. 2, No. 3, August 1984, pp. 181-197.

[MCKUSICK1996] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels and John S. Quarterman. The Design and Implementation of the 4.4 BSD Operating System. Addison-Wesley, April 30, 1996.

[MCKUSICK1999] Marshall Kirk McKusick and Gregory R. Granger. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. Proc. FREENIX Track: USENIX Annual Technical Conference, Monterey, CA, USA, June 6-11, 1999, pp. 1-17.

[MCKUSICK2002] Marshall Kirk McKusick. Running “fsck” in the Background. Proc. BSDCon 2002, February 11-14, 2002, San Francisco, CA, USA, pp. 55-64.

[MICROSOFT2002] Microsoft Corporation. Background Intelligent Transfer Service. Microsoft Windows Server Technical Article, August 2002.

[MINNICH1989] Ronald G. Minnich and David J. Farber. The Mether system: A distributed shared memory for SunOS 4.0. Proc. Summer USENIX Conference, Baltimore, MY, USA, June 12-16, 1989, pp. 51-60.

[MOGUL1990] Jeffrey C. Mogul and Stephen Deering. Path MTU Discovery. RFC 1191, November 1990.

[MOGUL1992] Jeffrey C. Mogul. Observing TCP Dynamics in Real Networks. Proc. ACM SIGCOMM, Baltimore, MY, USA, August 17-20, 1992, pp. 305-317.

[MOGUL1997] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. ACM Transactions on Computer Systems, Vol. 15, No. 3, August 1997, pp. 217-252.

[MOLANO1998] Anastasio Molano, Ragunathan Rajkumar and Kanaka Juvva. Dynamic Disk Bandwidth Management and Metadata Prefetching in a Realtime Filesystem. Proc. IEEE Euromicro Workshop on Realtime Systems, Berlin, Germany, June 17-1, 1998, pp. 203-213.

[MOSBERGER1996] David Mosberger and Larry L. Peterson. Making Paths Explicit in the Scout Operating System. Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA, USA, October 28-31, 1996, pp. 153-168.

[MOWRY1996] Todd C. Mowry, Angela K. Demke and Orran Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA, USA, October 28-31, 1996, pp. 3-17.

[MOWRY1998] Todd C. Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. ACM Transactions on Computer Systems, Vol. 16, No. 1, February 1998, pp. 55-92.

[MUMOLO1999] Enzo Mumolo. Prediction of Disk Arm Movements in Anticipation of Future Requests. Proc. IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), College Park, MD, USA, October 24-28, 1999, pp. 305-312.

[MUTKA1987] Matt W. Mutka and Miron Livny. Profiling Workstations' Available Capacity For Remote Execution. Proc. IFIP WG 7.3 Symposium on Computer Performance, Brussels, Belgium, December 7-9, 1987, pp. 529-544.

[MUTKA1991] Matt W. Mutka and Miron Livny. The available capacity of a privately owned workstation environment. Performance Evaluation, Vol. 12, 1991, pp. 269-284.

[NAHRSTEDT1996] Klara Nahrstedt, and Jonathan M. Smith. Design, Implementation and Experiences with the OMEGA End-point Architecture. IEEE Journal on Selected Areas in Communications (JSAC), Vol. 17, No. 7, September 1996, pp. 1263-1279.

[NARTEN1992] Thomas Narten and Raj Yavatkar. Remote Memory as a Resource in Distributed Systems. Proc. IEEE Workshop on Operating Systems, Key Biscane, FL, USA, April 23-24, 1992, pp. 132-136.

[NICHOLS1987] David A. Nichols. Using Idle Workstations in a Shared Computing Environment. Proc. ACM Symposium on Operating Systems Principles (SOSP), Austin, TX, USA, November 8-11, 1987, pp. 5-12.

[OZAWA1995] Toshihiro Ozawa, Yasunori Kimura and Shin'ichiro Nishizaki. Cache Miss Heuristics and Preloading Techniques for General-Purpose Programs. Proc. ACM International Symposium on Microarchitecture (MICRO), Ann Arbor, MI, USA, November 29 - December 1 1995, pp. 243-248.

[PADMANABHAN1996] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve World-Wide Web latency. ACM SIGCOMM Computer Communication Review, Vol. 27, No. 3, 1996, pp. 22-36.

[PADMANABHAN1998] Venkata N. Padmanabhan and Randy H. Katz. TCP Fast Start: A Technique For speeding Up Web Transfers. Proc. IEEE GLOBECOM Internet Mini-Conference, November 8-12, 1998, Sydney, Australia, pp. 41-46.

[PAPATHANASIOU2003] Athanasios E. Papathanasiou and Michael L. Scott. Energy Efficiency through Burstiness. To Appear IEEE Workshop on Mobile Computing Systems and Applications (WMCSA), Monterey, CA, USA, October 9-10, 2003.

[PARSA1999] Christina Parsa and J.J. Garcia-Luna-Aceves. Improving TCP Congestion Control Over Internets with Heterogeneous Transmission Media. Proc. IEEE International Conference on Network Protocols (ICNP), Toronto, Canada, October 31-November 3, 1999, pp. 213-221.

[PATTERSON1988] David A. Patterson, Garth A. Gibson and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). Proc. ACM SIGMOD International Conference on Management of Data, Chicago, IL, USA, June 1-3, 1988, pp. 109-116.

[PATTERSON1995] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky and Jim Zelenka. Informed Prefetching and Caching. Proc. ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain Resort, CO, USA, December 3-6, 1995, pp. 79-95.

[PIERCE1994] Jim Pierce and Trevor Mudge. The Effect of Speculative Execution on Cache Performance. Proc. Parallel Processing Symposium (IPPS), Cancun, Mexico, April 26-29, 1994, pp. 172-179.

[POPEK1981] Gerald Popek, Bruce Walker, Johanna Chow, David Edwards, Charles Kline, Gerald Rudisin, and Greg Thiel. LOCUS: A Network Transparent, High Reliability Distributed System. Proc. ACM Symposium on Operating Systems Principles (SOSP), Pacific Grove, CA, USA, December 14-16, 1981, pp. 169-177.

[POSIX1993] POSIX 1003.1b-1993. Portable Operating System Interface (POSIX) Part 1: System Application Program Interface Amendment 1: Realtime Extension [C Language], 1993.

[POSTEL1981] Jon Postel. DARPA Internet Protocol Specification. RFC 791, September 1981.

[POSTEL1983] Jon Postel. Discard Protocol. RFC 863, May 1983.

[POSTEL1998] Jon Postel. Private Communication. 1998.

[RASHID1988] Richard F. Rashid, Avadis Tevanian, Michael Young, David B. Golub, Robert V. Baron, David L. Black, William J. Bolosky, Jonathan Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. IEEE Transactions on Computers, Vol. 37, No. 8, 1988, pp. 896-907.

[REZNICK1993] Larry Reznick. Using cron and crontab. Sys Admin: The Journal for UNIX Systems Administrators, Vol. 2, No. 4, July/August 1993, pp. 29-34.

[RICHARDSON2003] Michael Richardson, and D. Hugh Redelmeier. Opportunistic Encryption using The Internet Key Exchange (IKE). Work In Progress (draft-richardson-ipsec-opportunistic-12.txt), June 2003.

[RIZZO1997] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. ACM SIGCOMM Computer Communication Review, Vol. 27, No. 1, January 1997 pp. 31-41.

[ROBERSON2002] Jeff Roberson. FreeBSD “prio” patch, February 2002.

[SCHIMMEL1994] Curt Schimmel. UNIX Systems for Modern Architectures. Addison-Wesley, 1994.

[SHALUNOV2001] Stanislav Shalunov and Benjamin Teitelbaum. QBone Scavenger Service (QBSS) Definition. Internet2 Technical Report, March 16, 2001.

[SPRUNT1988] Brinkley Sprunt, David Kirk and Lui Sha. Priority-Driven, Preemptive I/O Controllers for Realtime Systems. Proc. IEEE Annual International Symposium on Computer Architecture, Honolulu, HI, USA, May/June 1988, pp. 152-159.

[STANKOVIC1991] John A. Stankovic and Krithi Ramamritham. The Spring Kernel: A New Paradigm for Realtime Systems. IEEE Software, Vol. 8, No. 4, May 1991, pp. 62-72.

[SUGERMAN2001] Jeremy Sugerman, Ganesh Venkitachalam and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. Proc. USENIX Annual Technical Conference, Boston, MA, USA, June 25-30, 2001, pp. 1-14.

[SULLIVAN2000] David G. Sullivan and Margo I. Seltzer. Isolation with Flexibility: A Resource Management Framework for Central Servers. Proc. USENIX Annual Technical Conference, San Diego, CA, USA, June 19-23, 2000, pp. 337-350.

[TANDIARY1996] Fredy Tandiary, Suraj C. Kothari, Ashish Dixit, and E. Walter Anderson. Batrun: Utilizing Idle Workstations for Large-scale Computing. IEEE Parallel and Distributed Technology, Vol. 4, No. 2, 1996, pp. 41-48.

[THEIMER1985] Marvin M. Theimer, Keith A. Lantz and David R. Cheriton. Preemptable Remote Execution Facilities for the V-System. Proc. ACM Symposium on Operating Systems Principles (SOSP), Orcas Island, WA, USA, December 1985, pp. 2-12.

[THIBODEAU1998] Jan Thibodeau (editor). The Basic Guide to Frame Relay Networking. Frame Relay Forum, Fremont, CA, USA, 1998.

[THOMASIAN1998] Alexander Thomasian. Concurrency Control: Methods, Performance and Analysis. ACM Computing Surveys, Vol. 30, No. 1, March 1998, pp. 70-119.

[TOKUDA1990] Hideyuki Tokuda, Tatsuo Nakajima and Prithvi Rao. Realtime Mach: Towards a Predictable Realtime System. Proc. USENIX Mach Symposium, Burlington, VT, USA, October 4-5, 1990, pp. 73-82.

[TOUCH1991] Joseph D. Touch and David J. Farber. Memory-side driven anticipatory instruction transfer interface with processor-side instruction selection. U.S. Patent #5,353,419, University of Pennsylvania, September 1991. (Granted October 4, 1994.)

[TOUCH1992] Joseph D. Touch. Mirage: A Model for Latency in Communication. Ph.D. Dissertation, MS-CIS-92-42, DSL-11, Department of Computer and Information Science, University of Pennsylvania, January 1992.

[TOUCH1993] Joseph D. Touch. Parallel Communication. Proc. IEEE INFOCOM, San Francisco, CA, USA, March 28 - April 1, 1993, pp. 506-512.

[TOUCH1994] Joseph D. Touch and David J. Farber. An Experiment in Latency Reduction. Proc. IEEE INFOCOM, Toronto, Canada, June 12-16, 1994, pp. 175-181.

[TOUCH1998] Joseph D. Touch and Amy S. Hughes. The LSAM Proxy Cache – a Multicast Distributed Virtual Cache. Computer Networks and ISDN Systems, Vol. 30, No. 22-23, November 1998, pp. 2245-2252.

[TRAW1995] C. Brendan S. Traw and Jonathan M. Smith. Striping within the network subsystem. IEEE Network, Vol. 9, No. 4, July/August 1995, pp. 22-32.

[TRENT1995] Gene Trent and Mark Sake. WebSTONE: The First Generation in HTTP Server Benchmarking. Technical Report, Silicon Graphics, Inc., Mountain View, CA, February 1995.

[VARGHESE1997] George Varghese and Anthony Lauck. Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility. IEEE/ACM Transactions on Networking, Vol. 5, No. 6, 1997, pp. 824-834.

[VENKATARAMANI2002] Arun Venkataramani, Ravi Kokku and Mike Dahlin. TCP Nice: A Mechanism for Background Transfers. Proc. Symposium on Operating Systems Design and Implementation (OSDI), December 9-11, 2002, Boston, MA, USA.

[VISWESWARAIAH1997] Vikram Visweswaraiah and John Heidemann. Improving Restart of Idle TCP Connections. Technical Report 97-661, University of Southern California, November 1997.

[WALDSPURGER1995] Carl A. Waldspurger and William E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Management. Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, June 1995.

[WALLACE1998] Steven Wallace, Brad Calder and Dean M. Tullsen. Threaded Multiple Path Execution. Proc. ACM International Symposium on Computer Architecture (ISCA), June 27 - July 1, 1998, Barcelona, Spain, pp. 238-249.

[WORTHINGTON1994] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt. Scheduling Algorithms for Modern Disk Drives. Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Nashville, TN, USA, May 16-20, 1994, pp. 241-251.

[WYCKOFF1998] Peter Wyckoff, Theodore Johnson and Karpjoo Jeong. Finding Idle Periods on Networks of Workstations. Technical Report TR1998-761, Computer Science Department, New York University, March 1998.

[YAU1998] David K. Y. Yau and Simon S. Lam. Migrating Sockets – End System Support for Networking with Quality of Service Guarantees. IEEE/ACM Transactions on Networking, Vol. 6, No. 6, December 1998, pp. 700-716.

[YEE1993] Jenn-Jong Yee, Ming-Dong Feng and Chung-Kwong Yuen. Speculative Processing Mechanisms in a Parallel Lisp Machine: BIDDLE. Proc. Hawaii International Conference on System Sciences, Vol. 1, Kihei, HI, January 5-8, 1993, pp. 457-465.

[YU1994] Philip S. Yu, Kun-Lung Wu, Kwei-Jay Lin and Sang H. Son. On Realtime Databases: Concurrency Control and Scheduling. Proc. IEEE, Special Issue on Realtime Systems, January 1994, pp. 140-157.

[ZEC2003] Marko Zec and Miljenko Mikuc. Real-Time IP Network Simulation at Gigabit Data Rates. Proc. International Conference on Telecommunications (ConTEL), Zagreb, Croatia, June 11-13, 2003.

[ZHENG2003] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Currentcy: Unifying Policies for Resource Management. Proc. USENIX Annual Technical Conference, San Antonio, TX, USA, June 9-14, 2003, pp. 43-56.