FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

PLUGGING I/O RESOURCE LEAKS IN GENERAL PURPOSE REAL-TIME SYSTEMS

By

MARK J. STANOVICH

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Spring Semester, 2015

Copyright © 2015 Mark J. Stanovich. All Rights Reserved.

Mark J. Stanovich defended this dissertation on April 13, 2015.
The members of the supervisory committee were:

Theodore P. Baker
Professor Co-Directing Dissertation

An-I Andy Wang
Professor Co-Directing Dissertation

Emmanuel G. Collins
University Representative

Piyush Kumar
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the dissertation has been approved in accordance with university requirements.

To Mom, Dad, Lenore, and Jennifer

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisers Dr. Theodore Baker and Dr. Andy Wang. Their guidance, insight, and support have not only been one of the most significant reasons that I was able to complete this dissertation, but even more so have contributed to my development as a researcher.

My first meaningful encounter with Dr. Baker was in his kernel device drivers class. During this class is when I first gained a true appreciation for how little of computer science I truly understood. I am truly blessed to have had the opportunity to have been able to receive his teachings and guidance. Nearly every discussion with him has been enlightening. However, most of what I have learned from him was not from the discussions, but rather by observing him. The examples of how to approach problems and convey ideas has given me exceptional templates by which to develop my own processes. His candor is something that I have always admired and appreciated. Whether discussing an idea or reviewing my writing, he always expressed his honest opinion.

Dr. Andy Wang is the other key person directly responsible for me completing dissertation. He has taught me so much. In particular, his systematic methods have taught me how to approach any problem no matter how large or small. I am very grateful for his patience. Week after week I would show up with obstacles in my research and he would guide me to a plan that would eventually overcome them.

I would also like to thank the members of my dissertation committee. Dr. Piyush Kumar, in his algorithms class, provided me with one of my first glimpses into the field of computer science beyond just coding. I am also very fortunate to have Dr. Emmanuel Collins on my committee. Being an outside committee member, his questions and comments on the bigger picture have helped tremendously.

The members of the Operating Systems and Storage Research Group are owed a great deal of gratitude. The diversity of expertise and character provided a valuable asset to discuss ideas and develop a deeper understanding of my research. In particular, I would like to thank Chris Meyers and Sarah Diesburg for their numerous discussions.

Part of my time at Florida State was spent at Center for Advanced Power Systems (CAPS) and there are several people there that I would like to acknowledge, including Dr. Sanjeev Srivastava, Dr. Mischa Steurer, Dr. David Cartes, and Michael Sloderbeck. While this dissertation does not specifically describe research results during my time at CAPS, the experiences, knowledge, and skills have certainly contributed to the completion of this dissertation. I am very thankful for the opportunity work with such a great team of researchers.

Finally, I must express my deepest gratitude to God. This dissertation is only possible due to His overabundance of blessings throughout the journey. Thank you Jesus for this gift.

This dissertation contains content from the following publications:

[61] Mark Lewandowski, Mark J. Stanovich, Theodore P. Baker, Kartik Gopalan, and An-I Wang. "Modeling Device Driver Effects in Real-Time Schedulability Analysis: Study of a Network Driver," in *Proc. Of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 57–68, Apr. 2007.

[10] Theodore P. Baker, An-I Andy Wang, and Mark J. Stanovich. "Fitting Linux Device Drivers Into an Analyzable Scheduling Framework," in *Proc. of the 3rd Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pp. 1–9, July 2007

[107] Mark J. Stanovich, Theodore P. Baker, and An-I Andy Wang. "Throttling On-Disk Schedulers to Meet Soft-Real-Time Requirements," in *Proc. of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '08)*, pp. 331–341, Apr. 2008.

[109] Mark J. Stanovich, Theodore P. Baker, Andy An-I Wang, and M. González Harbour. "Defects of the POSIX Sporadic Server and How to Correct Them," in *Proc. of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 35–45, Apr. 2010.

[108] Mark J. Stanovich, Theodore P. Baker, and Andy An-I Wang. "Experience with Sporadic Server Scheduling in Linux: Theory vs. Practice," in *Proc. of the 13th Real-Time Linux Workshop (RTLWS)*, pp. 219–230, Oct. 2011.

Additional publications not directly related to this dissertation research:

- Sarah Diesburg, Christopher Meyers, Mark Stanovich, Michael Mitchell, Justin Marshall, Julia Gould, An-I Andy Wang, and Geoff Kuenning. "TrueErase: Per-File Secure Deletion for the Storage Data Path," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, pp. 439–448, Dec. 2012.
- M.J. Stanovich, I. Leonard, K. Sanjeev, M. Steurer, T.P. Roth, S. Jackson, and M. Bruce. "Development of a Smart-Grid Cyber-Physical Systems Testbed," in *Proc. of the 4th IEEE Conference on Innovative Smart Grid Technologies (ISGT)*, Feb. 2013.

- M.J. Stanovich, S.K. Srivastava, D.A. Cartes, and T.L. Bevis. “Multi-Agent Testbed for Emerging Power Systems,” in *Proc. of the IEEE Power and Energy Society General Meeting (PES-GM)*, July 2013.

Portions of this work were sponsored by the National Science Foundation under grant CNS-0509131. Opinions, findings, and conclusions or recommendations expressed in this document do not necessarily reflect the views of the NSF, FSU, or the U.S. government.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The I/O services implemented by general-purpose operating systems (GPOS) and commodity I/O devices are typically designed for high average-case performance without consideration of application-level timing constraints. Nevertheless, for economic reasons, such components are being used as building blocks for applications that are expected to meet timing constraints in order to function properly. A body of research on real-time (RT) theoretical analysis provides techniques for designing a system to meet timing constraints, however, the optimizations used for average-case performance in commodity I/O devices and GPOS do not lend themselves to such analysis. The most common approaches for dealing with this problem on general-purpose systems are to either disable optimizations or to forego applying any RT analysis. Disabling optimizations often results in poor performance. On the other hand, foregoing RT analysis means there is no assurance that the system will meet its timing constraints.

The thesis of this dissertation is that adapting GPOS I/O scheduling to balance high throughput and low latency is practical and necessary for broadening real-time support for guaranteeing a larger number and wider variety of application timing constraints. By capitalizing on average-case optimizations, a general purpose system is able to guarantee a wider spectrum of I/O timing constraints and increase system-wide performance, which includes applications that do not specify timing constraints. An observation underlying this thesis is that even though optimizations to improve average-case throughput may result in longer worst-case response times, there is a trade-off: poor average-case throughput can cause a system to develop a backlog that translates into missed deadlines.

This dissertation describes a journey to develop a new approach to extend real-time support in GPOS that provide I/O services. I/O services in a GPOS typically require both CPU and I/O device resources. The proper scheduling of these resources is critical to guarantee application timing constraints will be met. The contributions of this dissertation address the impact that scheduling I/O services has on throughput and latency. In the context of CPU resources, the overhead of switching between activities to achieve low latency is balanced with the reduction in processing throughput. In the context of I/O devices, throughput-improving optimizations offered by devices are leveraged, but are indirectly overridden when timing constraints might be in jeopardy. The additional levels of performance achievable by balancing throughput and latency translates into a larger number and wider variety of supported application timing constraints.

Contributions of this dissertation include the following: (1) techniques to measure and derive RT workload models for I/O devices and device drivers, (2) a throttling approach to indirectly control built-in hardware scheduling optimizations for meeting time constrained I/O service requirements (3) a modified aperiodic scheduling algorithm to reduce the effect of CPU context switching overhead incurred when providing I/O service, and (4) a novel scheduling algorithm and subsequent analysis to balance throughput and response time depending on requested RT I/O guarantees. These contributions are demonstrated by experiments with full-scale implementations using a GPOS (Linux) and commodity hardware.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Commercial-of-the-shelf (COTS) components are increasingly being used to create computer systems. Examples include hard disks used by Netflix to stream videos [81] and the Android OS [35] used as the primary system software for many cellular phones. Composing systems using COTS components has many benefits as compared to using custom-built components, including: shorter-time-to-market, lower cost, developer familiarity, and proven track record. The ability to reuse the same set of components with relatively few changes results in increased confidence that the intended system can be implemented without excessive, unexpected problems. As an example of reuse, the Linux kernel is used to build products from televisions [36] to space vehicles [27]. Such a large number of applications results in improved understanding of a particular COTS component's behavior and characteristics. Further, the large number of users provides a large experience base and allows for extensive testing and discovering the "bugs" that are almost certain to exist in any given component.

Many of these systems built from COTS components need to meet timing constraints for correct operation. These timing constraints are often in terms of explicitly stated deadlines by which some work must be accomplished. For instance, when talking on a phone, audio data packets must transferred before some deadline in order to provide expected, real-time communication. When deadlines are not met, failure in the form of choppy and often useless audio results.

An enormous amount of research has been performed over the past several decades, producing powerful techniques that can assist with designing a system to meet timing constraints [103]. In general, these techniques are applied to a given system by designing the system in terms of abstract, mathematical models ("RT models"). As long as the system adheres to the abstract models of the theory, even if the system has a potentially infinite number of states, it can be verified to have correct timing behavior.

Many COTS components are designed for average-case performance and do not lend themselves to abstract RT models. Both software and hardware often have behaviors that only loosely resemble existing abstract RT models. Often, the mismatch is due to unknowns of run-time system behavior. For instance, the time to service a hard disk I/O request depends on the state of its internal components (e.g., cache). In order to compensate for such unknowns, the typical approach is use RT analyses that are correct as long as the model's I/O service time is greater than or equal to any actual service time on the running system, referred to as the worst-case execution time (WCET). WCET values that are much larger than usual execution time values mean that the theoretical analysis can only guarantee deadlines for a small fraction of possible requests.

Average-case performance is often measured in terms of service over some arbitrarily large time interval. In most cases, performance may be close to the average, but there may be a few, infrequent time intervals of poor performance. Given that the abstract RT models only consider the worst-case performance values, the average-case performance is substantially higher than the performance able to be guaranteed by the RT analysis.

A WCET value can sometimes be reduced by disabling average-case optimizations. While disabling an optimization will likely reduce average-case performance, in some instances it may also reduce the WCET, thereby increasing RT performance. Consider a simplified, but real-world analogy where a bus travels around town picking up and dropping off passengers. Assume that the stops that bus makes are based on people calling in and requesting to be picked up at a stop closest to their current location. Further assume that each person only calls once per day and the maximum number of people that may call is known (it is a small town), but the exact time that any person may call or whether they will call in at all on a particular day is not known. Once the customer calls, the bus company can either answer the phone or can put them on a wait queue. However, once the company answers the phone the bus must commit to picking them up (the company does not want to be give the impression of preferential treatment). Also assume that the bus driver will always go to the pick-up or drop-off location that takes the least amount of time from his present location (trying to save fuel). Now given this strategy, say the bus company wants to guarantee some maximum length of time it will take to transport any passenger. Answering the phone when the bus is idle is generally not a bad idea (until we start talking about priority customers). Therefore, after answering the phone, the bus sets out to pick up the next passenger, now while the bus is out the phone rings. Should the operator answer the phone? In general, when should the phone be answered?

In a similar way, average-case performance hardware may have built-in schedulers that operate similarly to the bus driver and only service requests in what they perceive to be the most efficient manner. The most common approach for existing techniques is for the OS to disable the built-in scheduler by sending only one request at a time to the hardware and assume that the same scheduling decisions can be made in software. For meeting timing guarantees, the system software has more accurate knowledge of application-level timing constraints (e.g., deadlines). Therefore, in scholarly publications on real-time systems disabling the built-in scheduler of the I/O device is often considered a reasonable choice. However, inaccurate knowledge of device internals like the disk layout reduces the ability of system software to perform as well as a hardware device's built-in scheduler. Similarly, other average-case designs are often ignored by researchers in real-time systems theory assuming that a "better" implementation can be created. Every so often there are new OSs that promise much better real-time performance. However, developing and debugging an operating system, with all its device drivers, represents a huge investment. The knowledge and experience developed over many years in a "legacy" OS (e.g., Linux) is much more valuable than the supposed promises of a totally redesigned system.

As a consequence, the two most common approaches when building a system from commodity components is to either disable the optimizations in order to apply RT theoretical analyses or forego the use of the analysis. Disabling the optimization allows a closer fit to the abstract models, but often provides poor performance such that the application must be scaled back to fit. On the other hand, enabling these optimizations provides much better performance and can support many more functions on the same system, but the assurance that deadlines will be met is not as strong as if the RT theoretical analysis were used.

This dissertation shows how average-case designed I/O components and many associated optimizations can be used effectively with existing RT theoretical analysis.

1.2 Thesis

The dissertation supports the following thesis:

Adapting general-purpose operating system I/O scheduling to balance high throughput and low latency is practical and necessary for guaranteeing a larger number and wider variety of application timing constraints.

1.3 Scope of the Dissertation

The topic of RT I/O scheduling is considerably broad and include topics such as network scheduling, storage device scheduling (e.g., hard disk), and OS design just to name a few. The scope of this dissertation is limited to providing timely I/O service to applications run on a general-purpose operation systems using commodity hardware. In order to further expand on the scope and context of this research, the following describes several assumptions about the system that this research addresses.

- **Soft real-time systems:** A real-time system is considered *soft* real-time when a missed deadline (or more generally timing constraint) is unlikely to lead to a functional failure. The term soft real-time system is often contrasted with *hard* real-time systems. A missed deadline in a hard real-time system may result in partial or full loss of the system's intended function(s). Further, if the severity of the consequence of a missed deadline in a hard real-time system may conceivably result in loss of life or substantial monetary loss, the system is termed a *safety-critical* hard real-time system. Hard real-time systems typically run on specialized software and hardware and do not generally have the same type of COTS components addressed in this dissertation.

- **General-purpose OS:** Specialized RT operating systems are much better suited for hard real-time systems, but the availability of device drivers tends to be limited. Usually custom device drivers need to be written for such operating systems and therefore, do not allow the ease of composability considered in this research.

- **Built-in Features for Average-case Performance:** Many commodity components have built-in average-case performance features that are not easily modified or configured to meet arbitrary deadlines. For instance, hard disks are often shipped with a scheduler implemented in the component's firmware. This scheduler provides high average-case performance, but is not designed to meet arbitrary request deadlines. Further, changing the scheduler is considered to be impractical, since the manufacturer typically does not provide the necessary tools and sources to modify it. Certainly devices do exist that are very deterministic and/or can be easily configured for meeting arbitrary deadlines, however, many commodity components do not have either of these characteristics. Therefore, application of many RT scheduling techniques is not directly achievable.

- **Simple on-line scheduling algorithms:** General-purpose systems are typically built with online scheduling algorithms that have relatively simple logic for computing schedules. Online scheduling algorithms consume processing resources, shared with other real-time applications. Generally, the more complex the scheduling algorithm the more CPU time is required, resulting in less available CPU time. Further, implementing and ensuring a faithful representation of the theoretical scheduling algorithm to the implementation tends to become more error prone with complex online scheduling algorithms.

1.4 Contributions

This dissertation contributes to the field of real-time computing in the following ways:

(1) Provides insights into the gap between theory and implementation of RT I/O on general purpose systems, and introduces novel solutions to reduce the gap.

(2) Extends existing real-time theoretical scheduling analysis to take into account some often overlooked practical considerations.

(3) Demonstrates the application of RT theoretical techniques on a GPOS.

(4) Corrects errors in an existing real-time scheduling standards implemented by major operating systems.

CHAPTER 2

EXAMPLE APPLICATIONS

This chapter illustrates the problem addressed in this dissertation through the use of an example application. While the details of an actual production system will not be covered, the intention is to provide a high-level view to appreciate the problem and to demonstrate how the approach of balancing throughput and latency presented in this dissertation is practical and useful.

2.1 Introduction

Many real-world applications have deadlines. While some of these applications are considered hard real-time (e.g., avionic controls), where missing a deadline may result in serious (loss of function) failures, there are still many that are considered soft real time, where missing a deadline is not serious, but is at the same time not desired. If deadlines of a soft real-time system do occur, they should occur infrequently. A example of such an application is a multimedia application displaying a video. Loosely speaking, a sequence of images is expected to be displayed to the user within given time intervals. The display of each image has a specified time interval when it is expected to be changed from the previously displayed image. If one or more images are displayed too early or too late (outside the specified time interval), a video glitch will occur. While such video glitches are not catastrophic, the occurrence of does result in a reduction in the perceived performance of the application by the user.

2.2 Expectations and Requirements

The expectation of the application is to be provided timely access to resources in order to perform its function. Continuing with the multimedia application, the resources will often include reading data from the network or local hard disk, executing logic on the CPU for interacting with the user, and sending data to the local graphics processing unit to display images. The multimedia application is expected to execute in many different environments, with different competing applications. For instance, while a user is watching a video, a virus scanner may be running, updates may be being installed, or system backups may be being performed.

A system that results from piecing together a number of COTS components is often difficult to analyze. A broad range of design techniques are often used to help ensure that the composed system will function as expected. However, one design technique that is often overlooked are those of real-time theoretical scheduling. While such a pieced-together system may appear to work when tested, such appearances generally do not fully gauge whether deadlines will always be met. Much more confidence can be gained by leveraging real-time scheduling design techniques to provide more systematic analysis and therefore make it more likely that the final system will actually work. Ensuring the correct functioning of a given system is important because for mass-produced systems, where problems encountered after products are shipped and in use can be incur substantial, financial costs.

Certainly testing the functionality of a given system is important, but testing is often not able to explore the vastness of all possible system states. Again considering the multimedia example, it may be that a few movies are tested with a few examples of other applications also running on the system. However, the number of possible service orderings for using the CPU and I/O devices is often so large that testing only covers a very small fraction. Therefore, without verifying that these untested states do not result in missed deadlines, there is not much confidence that the system will actually work as expected when it is actually deployed.

The following describes a make-believe all-in-one small business server, illustrated in Figure 2.1, to provide some insight into the purpose of this dissertation research. The server is expected to provide video surveillance and run a small website. The server is intended to be economical and therefore uses COTS software and hardware. A few of the hardware components in the server are a hard disk, network interface card, and a CPU. The applications for the web server and video surveillance run on a general purpose operating system (e.g., Linux or Windows).

The video surveillance portion of the server is expected to receive data from one or more video cameras, analyze the video (e.g., intrusion detection), and locally store the video for playback. The video analysis is expected to be performed in a timely manner. For instance, detecting an intrusion too late may not be helpful as the perpetrators may have already exited. Analyzing the video in a timely manner relies on the general-purpose operating system (GPOS) passing the network I/O to the video analysis application. This dissertation provides techniques to properly design the GPOS to meet timing constraints of the network I/O. More specifically it addresses how to allocate and verify that CPU time for processing of network I/O in can be achieved without violating other application deadlines.

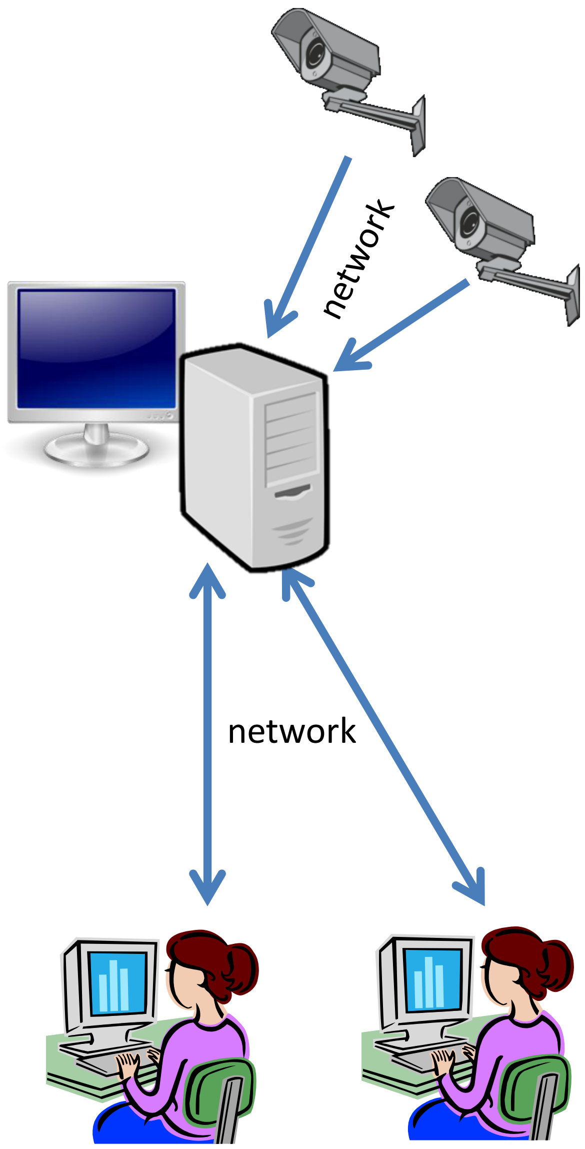


Figure 2.1: Illustration of an example all-in-one small-business server.

The server is also expected to perform timely hard disk I/O. The website, video surveillance recording, and playback all have timing constraints. Recording of video surveillance has timing constraints due to the limited amount of main memory used to buffer video frames. The website service has timing constraints for gathering data to present to a customer. The data is expected to be presented to the user in a timely manner. Suppose the website may also retrieve data (e.g, inventory) from a local database, which reside on the same hard disk.

It is desired to: (1) how best to schedule requests from different applications; and (2) determine, before the system is started, whether or not the system will always meet given timing constraints. That is, determine offline, whether a given configuration (e.g., hard disk model, scheduling algorithm) will meet the desired timing constraints. Generally speaking, (1) addresses how to make the most of a given resource and (2) addresses how to design the system to meet deadlines.

CHAPTER 3

SCHEDULING THEORY

Real-time system theory provides techniques to verify that real-world activities will complete within their associated timing constraints. A substantial amount of theoretical results exists from real-time systems research, much of which traces back to a seminal paper published by Liu and Layland in 1973 [67]. In general, RT theoretical techniques can be described as providing the ability to guarantee that a given abstract workload will be scheduled on a given abstract set of processing resources by a given scheduling algorithm in a way that will satisfy a given set of timing constraints. In this chapter a small portion of these theoretical results will be reviewed in order to explain common RT notation and terminology, as well as to provide background for understanding the difficulties encountered when attempting to apply RT theoretical techniques to provide RT I/O on general-purpose systems.

3.1 Workload Models

Workload models are designed to provide a mathematical representation of real-world activities. As an example, consider some hypothetical computations for injecting and igniting fuel in the cylinders of a gasoline engine .[1] Calculations for injection and ignition use sensor readings such as air temperature, altitude, throttle position, and others as inputs. Given the inputs, a sequence of processor instructions is used to compute actions (e.g., the time and the amount of fuel to inject). Execution of these processor instructions is considered *work* for the processor resource. The term typically used for one instance of processor work (e.g. one calculation) is a *job*. System functions such as fuel injection are performed over and over again, potentially resulting in an endless sequence of jobs. The sequence of jobs performing a particular system function is known as a *task* and commonly denoted by τ.

One way to visualize the work being performed on a given resource over time is through a Gantt chart. In Figure 3.1, each shaded block represents a job using a given resource for 5 time units. So,

[1]The gasoline engine used throughout this section is imaginary and used only as an illustrative analogy. The actual design of modern gasoline engines is at best more complicated than this and likely much different.

one job executes over the time interval between 0 and 5, another executes over the time interval between 25 and 30, etc. The amount of work performed by a given job will be referred to as the job's *execution time*. This is the amount of time the given job uses a resource. Note that all jobs of a given task may not have the same execution time. For instance, different code paths may be taken for a different sensor input values or one input may require fewer processor instructions, while another may require more.

For notational purposes, a given task is typically denoted by τ_i, where i is a generally a positive integer. Jobs of a task are denoted by j and the k^{th} job of task i is $j_{i,k}$. Therefore, a task τ_i can be considered as a sequence of jobs $j_{i,0}, j_{i,1}, ..., j_{i,k}, j_{i,k+1}, ...$. Although the execution time for jobs of given task may vary, the exact execution time of each job is often not known or able to be practically modeled. Instead, the largest execution time over all jobs of a given task, known as the worst-case execution time (WCET) is used.[2] For notational purposes, C_i is considered the WCET of τ_i.

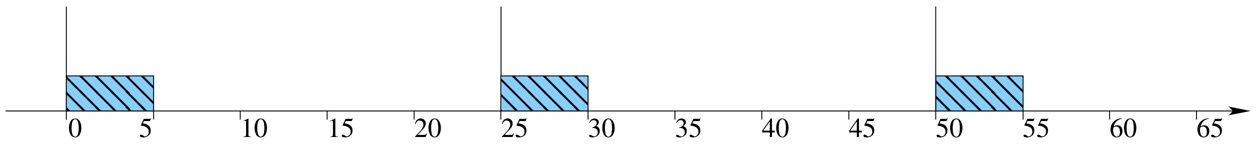


Figure 3.1: Gantt chart representing execution of work over time.

Each job has a *release time*, the earliest time instant when a job is allowed to begin execution. This release time may depend on data being available from sensors, another job being completed, or other reasons. The release time is not necessarily the time instant when the job begins execution, since it may be delayed due to factors such as another job using the resource.

Deadlines are used to explicitly define the time instant by which a job's work is to be completed. In an automobile engine, a deadline may be set so that the fuel must be injected before the piston reaches a certain position in its cylinder. At that position a spark will be produced and if the fuel is not present, no combustion will take place. The job's deadline is typically stated in either relative or absolute time. A relative deadline is specified as some number of time units from the release time of the job. On the other hand, absolute deadlines are in relation to time zero of the time-keeping

[2]An important observation to keep in mind throughout this dissertation is that one must critically consider the effects of using a model that differs from the actual implementation. For instance, one could naïvely assume that execution times shorter than a task's WCET cannot result in missed deadlines. However, for non-preemptive scheduling, it can be shown that if all jobs use the WCET no deadlines will be missed; where-as, jobs with execution times less than the WCET can cause missed deadlines.

clock (e.g., system start time). For example, a job with an absolute release time of $t = 12$ and relative deadline of 5 would have an absolute deadline of 17.

The periodic task model is one of the most commonly used abstract workload models used to model recurring job arrivals. With the periodic task model, each job is separated from the job immediately released prior by a constant length of time. For example, consider an engine running at a constant number of revolutions per minute (RPM). The calculation for the amount of fuel to inject must be performed some constant amount of time from the previous calculation. While the periodic task model does not directly apply if the RPMs of the engine change (slight modifications can take this into account and will be discussed shortly), many activities are inherently periodic or can be easily be configured to operate periodically (e.g., control algorithms). For notational purposes, the period of a periodic task is denoted by T_i meaning that the release of job, $j_{i,k+1}$, is T_i time units in the future from the release time of job $j_{i,k}$. That is, if $r_{i,k}$ denotes the release time of job $j_{i,k}$, $r_{i,k+1} - r_{i,k} = T_i$. Each job of a given periodic task has the same relative deadline, denoted by D_i. As such, the scheduling characteristics of a periodic task τ_i can be defined by the tuple (C_i, T_i, D_i).

The sporadic task model is a generalization of the periodic task model which treats the period of a task as only the lower bound on the inter-arrival time of jobs. That is, $r_{i,k+1} - r_{i,k} \geq T_i$. Figure 3.2 illustrates the sporadic task model. In this figure, $d_{i,k}$ represents the absolute deadline of job $j_{i,k}$. The sporadic task can be used to model the computations for an engine that runs at a varying RPMs, where the period of the sporadic task model would be the minimum time inter-release time of any two subsequent computations. For the computation of injecting fuel in the cylinder, this would be the period corresponding to the maximum RPM of the engine.

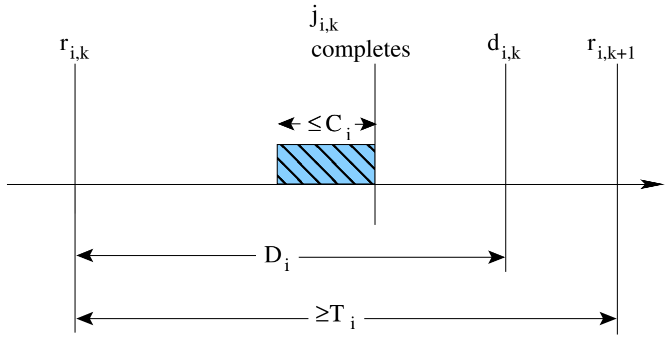


Figure 3.2: Illustration of timing constraints on a job of a sporadic task.

3.2 Scheduling

A scheduling algorithm specifies the time intervals any given job is able to use a given resource. RT scheduling algorithms can be broadly classified into two categories; static and dynamic. Dynamic schedulers wait until run time to determine the schedule. This generally means that the scheduling algorithm needs to be simpler than for static scheduling, since the scheduler competes with the scheduled jobs for use of the same computing resources. An advantage of a dynamic scheduler is that it has freedom to adapt the schedule to fluctuations in workloads, and can work with actual release times and execution times rather than a priori worst-case predictions. Therefore, a dynamic scheduler may succeed in meeting deadlines in situations where an "optimal" static schedule cannot find a feasible schedule. Dynamic scheduling also tends to be less âĂIJbrittleâĂİ than static scheduling, both in the sense of better tolerating variations in release times and execution times during the system operation, and also with respect to the impact of incremental software changes. Dynamic schedulers are typically preemptive, while static schedulers typically are not. Preemption allows greater scheduling freedom, and thereby allows scheduling a wider range of task systems. It also simplifies the theoretical analysis of schedulability, resulting in simpler and more robust schedulability tests. By far, the most prominent dynamic scheduling algorithms used in general-purpose operating systems are priority-based schedulers. Therefore, the remainder of this discussion will focus on them.

3.2.1 Static Schedulers

Static scheduling algorithms pre-compute (prior to the start of the system) the service order and time of each job's execution. One of the earliest static schedulers is known as the cyclic executive, in which a sequence of jobs is executed one after the other in a recurring fashion. The jobs are not preemptible and are run to completion. A cyclic executive is typically implemented as an infinite loop that deterministically releases each subsequent job after the previous job completes [68].

The cyclic executive model is simple to implement and validate, concurrency control is not necessary, and dependency constraints are taken into account by the scheduler. However, this model does have the drawback of being very inflexible [18, 69]. For instance, if the design of the system changes such that additional work is added to a given job, it is likely that the entire static schedule will need to be recomputed, affecting all the jobs. Such a change to the schedule will require additional testing and verification to ensure the original timing requirements are still met.

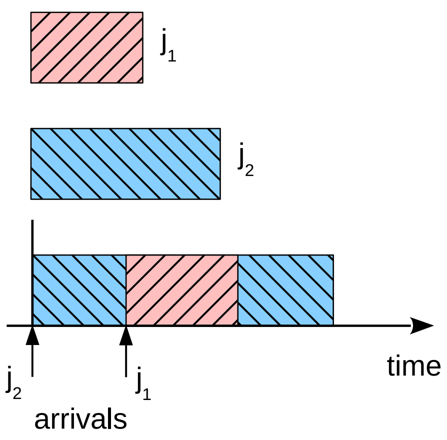
3.2.2 Dynamic Priority-Based Schedulers

Dynamic schedulers are much more adaptable to changes in workloads. Considering general-purpose systems, different designs often have different workloads. Therefore, it is desirable to keep the same verified scheduler in order to increase the reusability of existing verification that timing constraints will be met. As previously mentioned, static schedulers frequently change based on the workloads, however, dynamic schedulers are much more adaptable to changes in workloads and therefore, result in much less offline verification burden when a change in design modifies the workload. By far, the most prominent dynamic scheduling algorithms used in general-purpose systems are priority-based schedulers and therefore, the remainder of this discussion will focus on them.

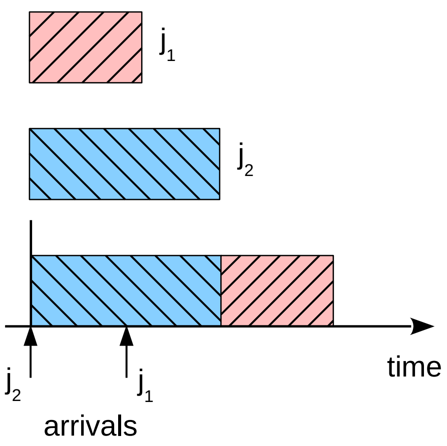
When multiple jobs contend to use a given resource (e.g., processor), order of service is resolved by allocating the resource to the job with the highest priority. A priority-based scheduler typically uses numeric priority values as the primary attribute to order access to a resource.

It is generally desired to provide the resource to the highest-priority job immediately. However, this is not always possible, either due to the scheduling algorithm or the given resource. This characteristic of being able to provide a resource immediately to a job is known as *preemption*. For instance, consider Figure 3.3a. Here we have two jobs, j_1 and j_2 sharing a resource. The job number also indicates the job's priority, and the lower the number indicates a higher priority. So, j_1 has higher priority than j_2. On the arrival of j_1, j_2 is stopped and execution of j_1 is started. This interruption of one job to start another job is known as a preemption. In this case, the higher priority job is able to preempt the lower priority job. If, j_1 is unable to preempt j_2 when it arrives as in Figure 3.3b, then j_2 is said to be non-preemptible. A non-preemptible job or resource means that once a job begins executing with the resource it will run to completion without interruption.

Preemption is not always desired. For instance, operations on data structures and devices may result in invalid behavior if accessed concurrently. To inhibit preemption some form of locking mechanism is typically used (e.g., monitors, mutexes, semaphores). However, preventing preemption can result in a violation of priority scheduling assumptions known as *priority inversion*. Priority



(a) Preemptive



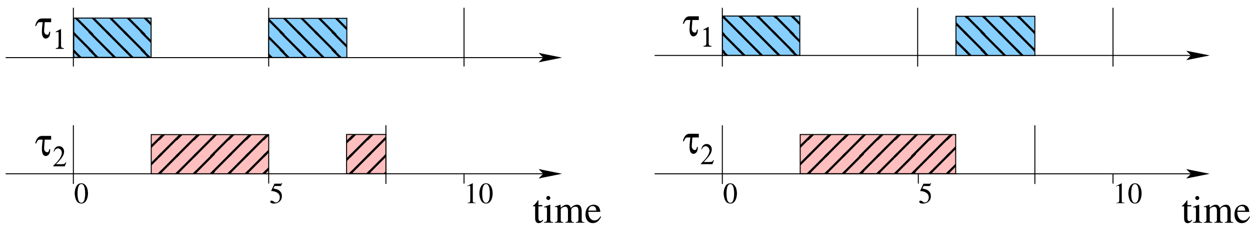
(b) Non-preemptive

Figure 3.3: Illustration of preemptive and non-preemptive scheduling.

inversion is a condition where a lower-priority task is executing, but at the same time a higher-priority task is not suspended but is also not executing. Using an example from [57], consider three tasks τ_1, τ_2, and τ_3, where the subscript also indicates the priority of the task. The smaller the numeric priority, the higher the task's priority. Further, consider a monitor M which τ_1 and τ_3 use for communication. Suppose τ_3 enters M and before τ_3 leaves M τ_2 preempts τ_3. While τ_2 is executing, τ_1 preempts τ_2 and τ_1 attempts to enter M, but is forced to wait (τ_3 is currently in M) and therefore is suspended. The next highest-priority task will be chosen to execute which is τ_2. Now, τ_2 will execute, effectively preventing τ_1 from executing, resulting in priority inversion.

Fixed-Task Priority Scheduling. Fixed-task-priority scheduling assigns priority values to tasks and all jobs of a given task are assigned the priority of their corresponding task. The assignment of priorities to tasks can be performed using a number of different policies. One widely known policy for assigning priorities for periodic tasks is what Liu and Layland termed rate-monotonic (RM) scheduling [67]. Using this scheduling policy, the shorter the task's period the higher the task's priority. One assumption of this policy is that the task's period is equal to its relative deadline. In order to generalize for tasks where the deadline may be less than the period, Audsley, et al. [6] introduced the deadline-monotonic scheduling policy. Rather than assigning priorities related to the period of the task, this approach schedules priorities according to the relative deadline of the task. Similar to RM scheduling, deadline-monotonic assigns a priority that is inversely proportional to the length of a task's deadline.

Fixed-Job Priority Scheduling. As with fixed-task priority scheduling the priority of a job does not change, however, with fixed-job priority scheduling, jobs of a given task may have different priority values. One of the most well-known fixed-job priority scheduling algorithms is earliest deadline first (EDF), in which the highest priority job is the job that has the earliest deadline. To illustrate the difference between fixed-job priority and fixed-task priority scheduling, consider Figure 3.4. τ_1 and τ_2 are periodic tasks assigned priority values using either EDF (dynamic) or RM (fixed) priorities. τ_1 has an execution time of 2 and period/deadline of 5. τ_2 has an execution time of 4 and period/deadline of 8. So, at time 0 the job of τ_1 has a higher priority than the job of τ_2 in both EDF and RM. At time 5, for RM, τ_1's next job again has higher priority, however, with EDF scheduling, τ_2's first job now has a higher priority than τ_1's second job, and hence the priority assignment for jobs of a single task may differ.



(a) fixed priority (RM) scheduling (b) dynamic priority (EDF) scheduling

Figure 3.4: Fixed vs. dynamic priority scheduling.

3.3 Schedulability

Schedulability analyses are used to mathematically verify that timing constraints of given abstract workload models, scheduled on a given set of abstract resources, using a given scheduling algorithm will always be met. In order to guarantee timing constraints the work to be performed on the system, the resources available to perform this work, and the schedule of access to the resources all must considered.

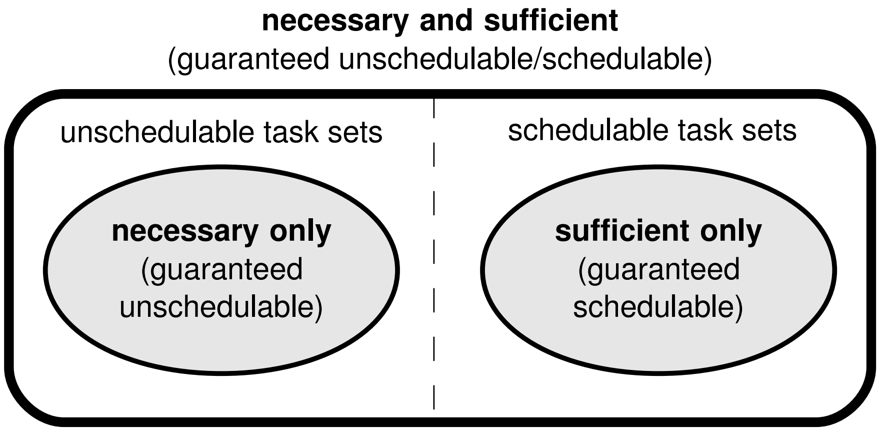


Figure 3.5: Types of guarantees made by various schedulability tests.

A schedulability test will typically report either a positive result, indicating that the task set is guaranteed to be schedulable, or a negative result, indicating the one or more jobs of the given task set may miss their deadlines. However, depending on the given schedulability test, the result may not be definite in either the positive or negative result. The terms *sufficient-only*, *necessary-only*, and *sufficient-and-necessary* are commonly used to distinguish between the different types of tests as described below and illustrated in Figure 3.5. A schedulability test where a positive result means the task set is guaranteed to be schedulable, but a negative result means that there is still a possibility that the task set is schedulable is termed a *sufficient-only* test. Similarly, a test where the negative result means that the task set is certainly unschedulable, but the positive result means there is still a possibility that the task set is unschedulable is a *necessary-only* test. Ideally one would always strive for tests that are *necessary-and-sufficient*, or exact, where a positive result means that all jobs are guaranteed to meet their deadlines and a negative result means that there is at least one scenario where a job may miss its deadline.

Liu and Layland published one of the first works on fixed-priority scheduling [67]. In their work, the critical instant theorem was formulated. The critical instant is the worst-case scenario for a given periodic task, which Liu and Layland showed occurs when the task is released with all tasks that have an equal or higher priority. This creates the most difficult scenario for the task to meet its deadline because the task will experience the largest amount of interference, thereby maximizing the job's response time.

Liu and Layland used the critical instant concept to develop the Critical Zone Theorem which states that for a given set of independent periodic tasks, if τ_i is released with all higher priority tasks and meets its first deadline, then τ_i will meet all future deadlines, regardless of the task release times [67]. This theorem was subsequently generalized to sporadic tasks. Using this theorem a necessary-and-sufficient test is developed by simulating all tasks at their critical instant to determine if they will meet their first deadline. If all tasks meet their first deadline, then the schedule is feasible. A naive implementation of this approach must consider all deadline and release points between the critical instant and the deadline of the lowest priority task. Therefore, for each task τ_i, one must consider $\lceil D_n/T_i \rceil$ such points, resulting in a complexity of $O(\sum_{i=0}^{n-1} \frac{D_n}{T_i})$ [103].

While schedulability analyses like the one above are useful for determining whether a particular task set is schedulable, it is sometimes preferable to think of task sets in more general terms. For instance, we may want to think of task parameters in terms of ranges rather than exact values. One approach that is particularly useful is known as maximum schedulable utilization, where the test to determine the schedulability of a task set is based on its total processor utilization. Utilization of a periodic or sporadic task is the fraction of processor time that the task can demand from a resource. Utilization is calculated by dividing the computation time by the period, $U_i = \frac{C_i}{T_i}$. The utilization of the task set, or total utilization, is then the sum of utilization of the individual tasks in the set, $U_{sum} = \sum_{i=0}^{n-1} U_i$, where n is the number of tasks in the set. Now to determine whether a task set is schedulable, one need only compare the utilization of the task set with that of the maximum schedulable utilization. As long as total utilization of the task set is less than maximum schedulable utilization then the task set is schedulable.

The maximum schedulable utilization varies depending on the scheduling policy. Considering a uniprocessor system with preemptive scheduling and periodic tasks assigned priorities according to the RM scheduling policy, the maximum schedulable utilization is $n(2^{\frac{1}{n}} - 1)$, and referred to as the RM utilization bound (U_{RM}) [67]. As long as $U_{sum} \leq n(2^{\frac{1}{n}} - 1)$ the tasks are guaranteed to always meet their deadlines. This RM utilization bound test is sufficient, but not necessary (failure of the test does not mean the task set is necessary unschedulable). Therefore, a task set satisfying the RM utilization test will always be schedulable, but task sets with higher utilization cannot be ensured schedulability.

While the RM utilization bound cannot guarantee any task sets above U_{RM}, one particularly useful class of task sets which can guarantee higher utilizations are those whose task periods are harmonic. These task sets can be guaranteed for utilizations up to 100% [103].

Preemptive EDF is another commonly used scheduling algorithm. The schedulability utilization bound provided for this scheduling policy is 100% on a uniprocessor [67]. This means that as long as the utilization of the task set does not exceed 100% then the task set is guaranteed to be schedulable. In fact, for a uniprocessor, the EDF scheduling algorithm is optimal, in the sense that if any feasible schedule exists than EDF can also produce a feasible schedule.

Many other scheduling algorithms and analyses exist to provide guarantees of meeting deadlines. This is especially true for the area of multiprocessor scheduling. However, the basic principles are essentially the same, namely, given a workload model and scheduling algorithm, a schedulability test is used determine whether timing constraints of a given system will be met.

CHAPTER 4

IMPLEMENTATION OF CPU TASKS

The OS's CPU scheduler allocates a system's processing unit (i.e., CPU) resource to individual threads of execution for lengths of time. A *thread of execution*, commonly just referred to as a thread, is a sequence of instructions executing on a processor (e.g., computation of an algorithm). In general, a running application is implemented by one or more threads. On a typical, single processor system, many threads coexist; however, a processing unit may be allocated to only one thread at a time.[1] Therefore, intervals of time on a processing unit are generally rotated among the threads contending for (i.e., requesting) processor usage. Deciding which of the contending threads and the length of the time interval is based on the scheduler's *scheduling algorithm*. Figure 4.1 provides an illustration of several threads (applications) contending for processor time. Since only contending threads are allocated processor time, the following will discuss typical thread states in terms of requesting and allocation of processor time.

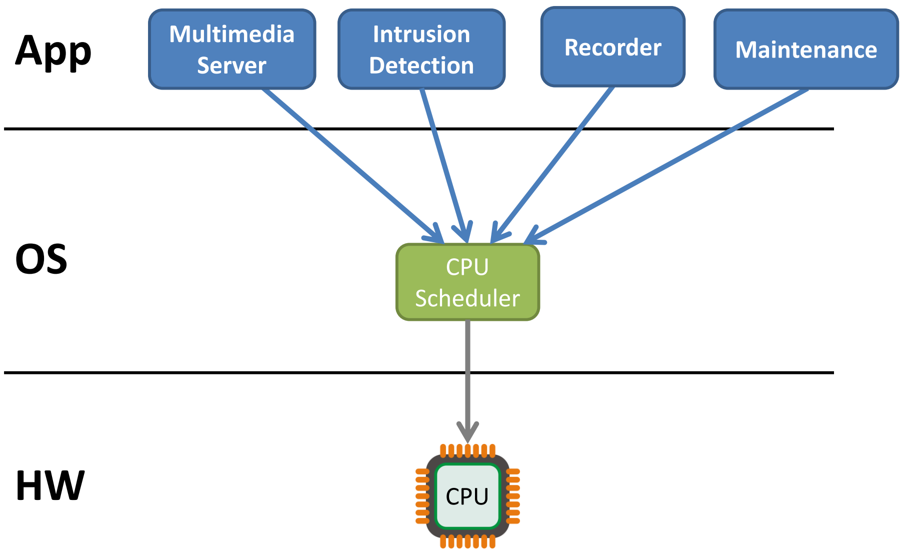


Figure 4.1: Illustration of threads (applications) contending for processor time.

[1]For ease of explanation, the discussion in this section is limited to a single processor system with one processing core, no hyper-threading, and energy optimizations such as dynamic voltage scaling are disabled, unless explicitly stated otherwise.

4.1 Contention

In terms of processor usage, a thread is generally considered to be in one of three states: blocked, ready, or running. Figure 4.2 illustrates the valid transitions from one state to another. To explain these states, consider a thread, denoted by τ_1, in the ready state. A thread in the ready state means that it is not executing on the processor but is requesting to execute a sequence of instructions on the processor. In the ready state, the thread is held in a “ready queue” and the thread is considered *runnable*. Suppose at some time instant, the scheduler decides that τ_1 should be allocated time on the processor. At that point, the scheduler performs what is known as a *context switch*, which is the operation of pausing the currently executing thread and resuming execution of another thread on the processor. For the following discussion, the thread that is being removed (paused) from the processor is referred to as the outgoing thread and the thread that is being allocated processor time (resumed) is referred to as the incoming thread. One of the first steps in a context switch involves saving all the information or context that will be needed to later resume the outgoing thread. This information must be saved since the incoming thread is likely to overwrite much of the context of the outgoing thread and would otherwise be lost. Next the incoming thread’s context will be restored to its original state when it was previously paused. At this point, processor control will be turned over to the incoming thread and the thread is considered to be in the *running* state.

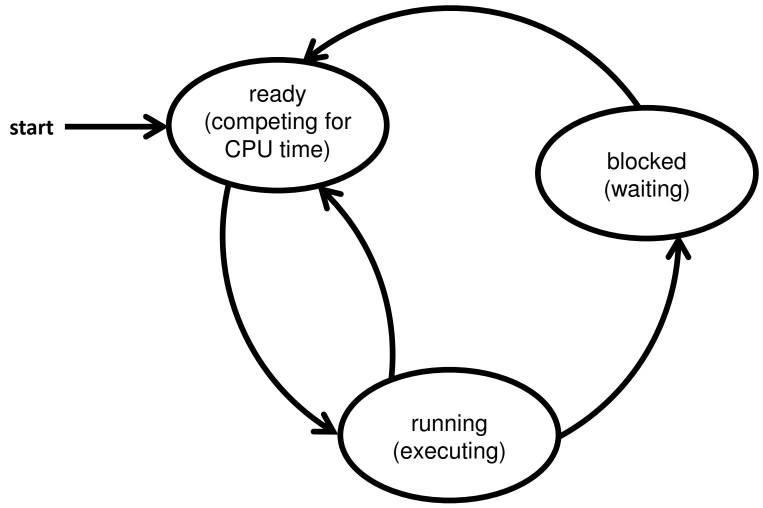


Figure 4.2: State diagram of a thread.

A thread in the running state may request some service, such as, reading from a file, sending a network packet, etc. Suppose the request cannot be fulfilled immediately and the thread has no further instruction to execute until the request is completed. In this case, the thread will transition to what is known as the *blocked* state. Once in the blocked state, the thread will not contend with other threads for processor time until the request completes.

4.2 Regaining Control

The scheduler is a component of the part of the OS known as the kernel and is implemented as set of instructions that is run on the processor. As a set of instructions, the scheduler is not any different from any other thread. Therefore, a logical question may arise as to how the scheduler gets scheduled.

Generally, two mechanisms are used to execute the scheduler thread: through calls by the currently executing thread to the OS kernel, and through hardware interrupts. A given thread commonly calls the kernel for services such as input or output (I/O) operations or to pause its execution until a future point in time (sleep). When the currently executing thread calls the kernel for any service, the scheduler code is usually executed as part of the kernel call. Forced execution of the scheduler code is also provided by hardware interrupts and is often needed as an alternative mechanism to execute the scheduler thread (e.g., if the currently executing thread does not call into the kernel when the scheduler needs to run).

An interrupt is a signal to the CPU to perform a context switch and execute a predefined set of instructions called an interrupt service routine (ISR). Interrupts are used to notify the CPU that some event has occurred. In the case of the scheduler, before completing a context switch, the scheduler will typically set a hardware timer to generate an interrupt some time instant in the future. Therefore, even if the currently executing thread does not call into the kernel, the scheduler thread is still able to execute and manipulate the time intervals that threads use CPU.

Hardware timers are typically capable of at least two modes of operation; periodic and one-shot. In periodic mode, the timer sends an interrupt repetitively after every fixed-length of time, specified by the scheduler, elapses. Figure 4.3 shows an example of a timer in periodic mode. In the figure, the periodic timer interrupt allows the scheduler to switch between threads A and B. The other timer interrupt mode is referred to as one-shot mode. In one-shot mode, the timer is set to arrive some specified time in the future, but unlike periodic mode, once the timer interrupt is issued, the scheduler must reset the timer for another interrupt.

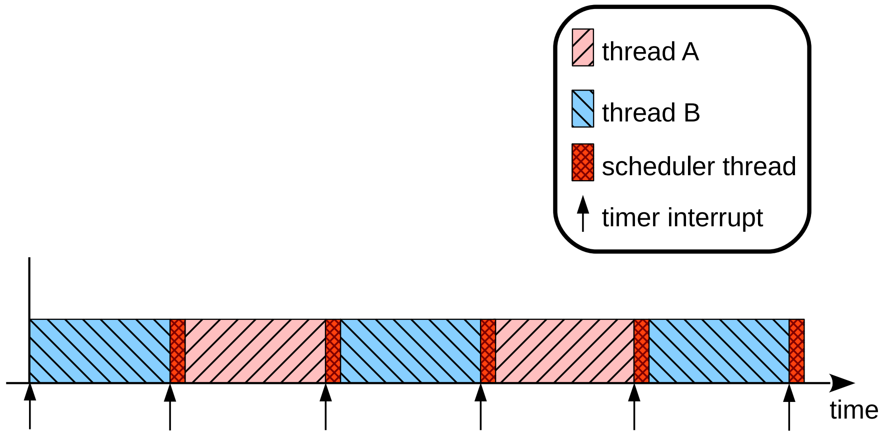


Figure 4.3: Periodic timer interrupt.

4.3 Considerations for Real-Time Implementation

The guarantees provided by real-time theoretical analysis require the implemented system to correctly match the theoretical models. Using appropriate real-time scheduling analysis provides a priori guarantees that timing constraints of a set of conceptual activities (e.g. threads) will be met. However, if the assumptions of the theoretical models do not hold true for the implemented system, the guarantees made by the scheduling analysis may no longer be valid. Whether a system can support a given theoretical model relies on the system's ability to control the behavior of tasks and to properly communicate timing parameters. Further, one must be able to extract timing information for implemented tasks (e.g., WCET).

4.3.1 Conveyance of Task/Scheduling Policy Semantics

For an implemented system to adhere to a given theoretical model, one must be able to convey the characteristics of this model to the implemented system. To perform this in a GPOS, it is common to provide a set of system interfaces (calls to the kernel) that provide the system with the task's parameters.

For example, consider the periodic task model scheduled with fixed-priority preemptive scheduling. Each task releases a job periodically and competes at its specified priority level until the job is completed. In a typical system, a task is implemented as a thread. Therefore, the thread should inform the OS scheduler that of its priority so that it can be scheduled in the same manner as the theoretical assumptions.

Many systems adhere to the POSIX operating systems standards [38], which provide commonly used real-time facilities to allow for implementation of a periodic task model scheduled using fixed-priority preemptive scheduling. These interfaces include functions for setting a fixed priority to a thread and allowing a thread to self-suspend when a job is completed. Thereby, allowing one to map a given task model to the implementation.

These types of interfaces are critical for applications to convey their intentions and constraints to the system. They inform the OS of the abstract model parameters in order for the OS scheduler to make decisions that match the theoretical scheduling algorithm. Lacking this information, the OS may make improper decisions, resulting in missed deadlines.

Priorities. Priorities are one of the most common parameters used to choose which thread, out of all runnable threads, is to use the processor. For scheduling non-real-time threads, one reasonable policy is to allow each thread to make similar progress. Assuming that all available threads started at approximately the same time, the scheduler may attempt to provide fairness among the runnable threads by choosing the one that has received the least amount of execution time in the recent past. However, providing fairness between all threads is not appropriate when one thread is more urgent than others.[2]

Basic priority scheduler implementations set the highest priority thread to occupy the processor until it suspends itself. This means that one thread can monopolize the CPU resource, causing the system to be unresponsive to other lower priority, runnable threads. Monopolizing the processor is not necessarily ill-behaved since the theoretical scheduling algorithm may indeed do the same thing. However, a high-priority thread that consumes more processor time than its theoretical counterpart may then cause the system to fail. For example, a misbehaving higher priority thread may result in lower priority threads from missing their deadlines. Therefore, it is important to the correct functioning of the system that implemented threads are programmed to release the processor (i.e.,

[2]It should be noted that in the following discussion priorities are used as a means to meet deadlines and not to convey importance of a given task to the correct functioning of the system as a whole. The relative importance a given task to the functioning of the entire system is commonly referred to as its criticality and is the area of research known as mixed criticality systems. Such systems are outside the scope of this dissertation.

self-suspend) in accordance with the theoretical model. In order to prevent a misbehaving task from causing other jobs to miss their deadlines, other scheduling algorithms, such as aperiodic servers can be used and will be discussed later in this dissertation.

Threads that have the same priority are often scheduled based on a *fifo* or *round-robin* scheduling policy. With a fifo scheduling policy, the thread that requested the CPU first will be allocated the CPU. Alternatively, the round-robin scheduling policy schedules each thread with the same priority for a fixed amount of time known as a time slice. All threads at a given priority value will receive one time slice before any thread of that level receives additional time slices.

Priority Spaces. When a device wishes to inform the CPU of some event, the device will send an interrupt signal to the processor, resulting in the execution of an interrupt handler. The interrupt handler is executed immediately without consulting the system's scheduler, creating two separate priority spaces: the hardware interrupt priority space and the OS scheduler's priority space. The hardware interrupt scheduler generally will preempt any OS scheduled thread regardless of priority value. This means that any OS schedulable thread may be preempted by a hardware interrupt handler. The fact that all interrupts have higher priority than all OS schedulable threads, must be modeled as such in the theoretical analysis. The common way to view the interrupts is as interference for other executing threads, which essentially increase the WCET of all tasks. As long as interrupt handlers require short lengths of time when compared to the time consumed by other jobs and do not occur too frequently, the increase in WCET does not have much impact on the ability of the system guarantee deadlines for the same workload. However, the more code that runs at interrupt priority the greater the amount of interference an OS schedulable thread may experience, potentially causing a reduction in the number of tasks that the theoretical analysis can guarantee to meet deadlines.

4.3.2 Processor Time Requirements and Accounting

The validity of schedulability analysis techniques depends on there being an accurate mapping of usage of the system's processor in terms of time to the given workload in the theoretical model. During execution of the system, all execution time must stay within the time bounds of the model. For example, considering the periodic task workload model, the time used for a particular job executing on the processor should not exceed its theoretical WCET. The proper accounting of all the time on a system is difficult. This section will cover some of the more common problems that hinder a system from achieving proper time accounting.

Variabilities in WCET. Nearly all task abstraction requires a WCET for any task. To determine the WCET of a task, one approach would be to enumerate all possible code paths that a task may take and use the time associated with the longest execution time path as the WCET. In a simple system such as a cyclic executive running on a simple microcontroller, this approach may work, but using a GPOS, this WCET would unlikely reflect the true WCET since tasks on such systems could have additional complexities such as context switching, caching, blocking due to I/O operations, and so on. The following will discuss some common causes for variabilities in a task's WCET.

The number of instructions, the speed to execute these instructions, caching, processor optimizations, etc. can introduce extremely large variabilities in the time to execute a sequence of instructions (i.e., code). As processors become increasingly complicated, the difficulty in determining accurate WCETs also becomes more complicated. Many difficulties arise from instruction-level parallelism in the processor pipeline, caching, branch prediction, etc. Such optimizations make it difficult to accurately determine a WCET parameter for a given implemented task.

There are generally three methods used to determine the WCET [82, 118]: compiler techniques [4, 41], simulators [79], and measurement techniques [61]. These methods can be effectively used together to take advantage of the strengths of each. For example, RapiTime [70, 71], a commercially available WCET analysis tool, combines measurement and static analysis. Static analysis is used to determine the overall structure of the code, and measurement techniques are used to establish the WCETs for sections of code on an actual processor.

Theoretical analysis generally relies on the WCET of one task not being affected by another task. In practice, such an assumption of the effect on WCET is typically not valid. For instance, contention for memory bus access and caching effects from a context switch may result in an increase in a job's execution time over the WCET experienced if executing without other tasks also executing on the system. In order to correctly model such effects in the theoretical model, the WCET can often be increased appropriately. As the trend toward more processors per system continues memory bus contention has become a very challenging problem. What processes are concurrently accessing which regions of memory can greatly affect the time to complete an activity. Further, processes are not the only entities competing for memory accesses; peripheral devices also access memory and therefore increase WCETs [85, 101].

Context switch overhead is typically small compared to the intervals of time a thread executes on a processor. However, if context switches occur often enough, this overhead becomes significant and must be accounted for in the analysis. Consider a job-level fixed-priority system where jobs cannot self suspend. If the time to perform a context switch is denoted as CS, then one needs to add $2 \cdot CS$ to the WCET of each job of a task [68]. The reason is that each job can preempt at most one other job, and each job can incur at most two context switches: one when starting and one at its completion. Similar reasoning can be used to allow for self-suspending jobs where each self suspension adds two additional context switches. Therefore, if S_i is the maximum number of self-suspensions per job for task i, then the WCET should be increased by $2(S_i + 1) \cdot CS$ [68].

Context switching consumes CPU time and should be included in the theoretical analyses. In order to do so, one must determine the time the time to perform a context switch. Ousterhout's empirical method [83] measures two processes communicating through a pipe. A process will create a pipe and fork off a child. Then the child and parent will switch between one and other each repeatedly performing a read and a write on the created pipe. Doing this some number of times provides an estimate on the cost of performing a context switch.

Ousterhout's method not only includes the cost of a context switch but also the cost of a read and a write system call on the pipe which in itself can contribute a significant amount of time. To factor out this time, McVoy and Staelin [76], measured the time of a single process performing the same number of write and read sequences on the pipe as performed by both processes previously. This measured time of only the system calls are subtracted from the time measured via Ousterhout's method, thereby leaving only the cost of the context switches. This method is implemented in the benchmarking tool, lmbench [75].

Note that context switch may be considered to include a deferred component, which is likely not measured in the above techniques. For instance, reloading the cache and translation lookaside buffer (TLB) often varies significantly depending on the size of the "working set" of the thread at the time it was suspended. This is not all incurred at once, but incrementally, as the thread re-references cache lines and page table entries.

Scheduling Overhead. The OS scheduler uses processor time and as such must be appropriately taken into account in the theoretical model. Katcher et al. [49] describe two types of scheduling interrupts, timer-driven and event-driven.

Tick scheduling [17] occurs when the timer periodically sends an interrupt to the processor. The interrupt handler then invokes the scheduler. From here, the scheduler will update the run queue by determining which tasks are available to execute. Any task that has release times at or before the current time will be put in the run queue and able to compete for the CPU at its given priority level. Performing these scheduling functions consumes CPU time which should be considered in the schedulability analysis. Overlooking system code called from a timer can be detrimental to the schedulability of a system because timer handlers can preempt any thread, regardless of the thread's priority or deadline.

System Workloads. A GPOS generally performs "housekeeping and bookkeeping" work to ease application development. This GPOS work contributes to the proper operation of the system, but at the same time consumes CPU time that reduces the total amount of CPU time available for applications to execute. Further, since GPOS work may not be the result of any particular task, it may not be included in any of the tasks' execution times and left out of the theoretical workload model. Without properly accounting for the CPU time of the GPOS in the abstract model, the GPOS work can unexpectedly "steal" execution time from jobs, thereby causing missed deadlines.

4.3.3 Temporal Control

Temporal control ensures that the enforcement mechanisms in the implementation correctly adhere to the real-time models used in the analysis. For the processor, this includes the system's ability to allocate the processor to a given activity in a timely manner. For example, when a job with a higher priority than that of the one currently executing on the processor is released (arrives), the preemptive scheduling model says the system should provide the processor to the higher priority job immediately. However, in practice *immediately* generally does not mean the same time instant since other considerations such as hardware latencies (e.g., context switch) and non-preemptible sections occur.

Non-preemptible Sections. Another common problem in real-world systems is that of non-preemptible sections. A non-preemptible section is a fragment of code that must complete execution before the processor may be given to another thread. Clearly, a non-preemptible section that is sufficiently long can cause a real-time task to miss its execution time window. While accounting for non-preemptible sections in the schedulability analysis is necessary for guaranteeing timing guarantees, it is generally preferable to minimize the length of non-preemptible sections. Intuitively, the longer the length of the non-preemptible sections, the fewer task sets can be guaranteed to meet deadlines.

Isolation. As mentioned previously, accurate WCETs can be extremely difficult to determine for general-purpose systems, therefore, WCETs tend to be estimates that are generally true. If a given job overruns its modeled WCET, it may cause one or more tasks to miss their deadlines. Rather than all tasks missing their deadlines, it is generally preferable to isolate the misbehaving task from other tasks on the system. This property is known as *temporal isolation*. In order to achieve temporal isolation a common approach is to use aperiodic server scheduling algorithms that limit a job/task's processor time to the estimated WCET.

4.4 Implementing Simple Periodic Tasks

A periodic task could be implemented as a thread with an assigned priority (assuming fixed-task priority) that periodically executes a sequence of instructions on the processor. Each periodic execution of the thread corresponds to the release of a subsequent job of the abstract task. The thread executes the sequence of instructions and upon completion of the sequence of instructions, the thread informs the scheduler to suspend the thread until the release time of the subsequent job. The thread requests a wake-up at the beginning of the next period.

The release time instant of a job is likely not the time instant the job begins executing its instructions on the processor. In an actual implementation instantaneous execution of a given job is not always possible. For instance, consider a thread that is the implementation of a periodic task denoted by τ_1. The common implementation on a general-purpose system to set a hardware timer to issue an interrupt at the thread's requested wake-up (assuming upon wake-up the thread would have the highest priority). In order to start execution of τ_1, the GPOS first must use the processor to execute its logic and bookkeeping and finally perform a context switch to the thread. The time from the requested start time and the actual start time is often considered *release time jitter* and results in a deviation from theoretical system. Delays due to release time jitter can be accounted for in the theoretical analysis as a blocking term. However, adding blocking terms to the theoretical system is generally not desired as it tends to reduce the total number of schedulable task sets. A simple example of the reduction in schedulability would be a periodic task with a deadline equal to its WCET. With release time jitter, the given periodic task may not start execution immediately upon release and therefore in the worst-case miss its deadline.

The general trend for GPOSs is to reduce the length and number of non-preemptible sections that result in release time jitter. For instance, many of the real-time modifications to the Linux kernel over the years have been devoted to reducing the number and size of non-preemptible sections that are within the kernel's code [60, 65, 74].

CHAPTER 5

MODELING PROCESSOR ALLOCATION EFFECTS OF I/O SERVICES

This chapter describes one of my research contributions for modeling I/O-service processor time usage on a GPOS in order to apply schedulability analysis. The specific contributions is an approach to derive theoretical load-bound functions for I/O service using empirical measurements.

A large portion of this chapter's results was the outcome of a collaborative effort amongst Mark Lewandowski, Theodore Baker, Kartik Gopalan, Andy Wang, and me. The results were published in [10, 61].

5.1 Introduction

This chapter explores the difficulties and provides a solution for modeling I/O-service processor time allocation into real-time schedulability analysis. While I/O services often utilize peripheral I/O devices, they generally also require some amount of processor (i.e., CPU) time to function correctly. Given that allocation of processor time to I/O services can delay real-time applications, the effects of I/O services must be considered to verify deadlines are not missed. That is, the timing guarantees resulting from theoretical schedulability analysis techniques are largely contingent on an accurate system model to verify that processor time will always be scheduled in such a way that deadlines will not be missed. On a GPOS, processor time allocated to I/O services can result in delaying real-time applications from obtaining needed processor time in order to meet their deadlines. Therefore, such I/O processor time must be included for schedulability analysis results to be valid on an implemented system. In particular, the allocation of processor time for I/O services tends to be scheduled in a way that does not allow a straightforward mapping to commonly used abstract workload models.

The I/O frameworks on GPOSs have often evolved over time with a goal of maximizing average-case performance. However, average-case and real-time performance are often at odds with one other. While average-case is generally concerned with unspecified, relatively long (i.e., seconds) time intervals, real-time designs are concerned with specific, relatively short (i.e., 100's of msecs or shorter) time intervals. Therefore, many expected system properties and settings of conventional real-time scheduling theory are not available. Thus, attempts to fit I/O-processor allocations into a real-time theoretical framework tends to be awkward, resulting in less than ideal timing guarantee results from the theoretical analyses.

Designing a new GPOS is one possible approach for creating a "better" environment for applying existing real-time theoretical techniques, but may be impractical. Well-established GPOSs have generally taken a considerable amount of time to achieve a system that functions reasonably well. Through years of testing and revisions, such GPOSs have developed a framework that has been demonstrated to work well for many applications. Further, a substantial amount of logic has been encoded into these systems in the form of device driver code. Therefore, one could envision that building a GPOS from scratch would likely take significant time and resources.

The approach taken in this chapter is to keep the I/O framework of existing GPOSs intact, but develop a method to derive a real-time workload model that accurately characterizes the I/O-service processor time. The derived model can then be used with existing schedulability analysis results to verify a system's timing constraints.

The outline for the remainder of the chapter is to discuss the general design for providing I/O service in a GPOS, with particular focus on processor usage, in Section 5.2. The following Section 5.3 discusses theoretical schedulability analysis and introduces a demand-based model of processor usage termed load bound. Section 5.4 describes our novel technique to derive such a load bound model that represents a network I/O service. Section 5.5 demonstrates the applicability of our approach by empirically deriving a load bound function for the network I/O service on a real system. Additionally, some brief comments are given about effect of varying the I/O-service processor thread's priority in terms of interference for other tasks and I/O quality. Lastly, Section 5.6 concludes the chapter.

5.2 Device Drivers

A GPOS commonly utilizes peripheral hardware devices to perform I/O operations such as sending and receiving network packets, accessing storage devices, displaying video, etc. Using such I/O devices requires detailed logic to convey the desired I/O operations to a given device (e.g., communication protocol). In a GPOS, *device drivers* take-on the role of encapsulating the logic for managing and communicating with specific hardware devices in order to fulfill application I/O requests [47]. More specifically, device drivers are software instructions for configuration, initialization, and general administration of the hardware devices, as well as, the translation of standardized OS commands into hardware-specific commands. Although, I/O functions can often be generically defined, such as sending a network packet, the actual hardware implementation and correct interaction with a given hardware device can vary substantially (e.g., wireless and wired network hardware). Therefore, in order to reduce the complexity of such differences when composing a system, the hardware-specific interfacing details are encapsulated into device drivers. A much more modular design can be achieved by separating out the I/O services provided (e.g., sending network packets) from specific hardware commands. Therefore, composing a system with different hardware components is achieved without requiring changes to the core OS and application code. Figure 5.1 illustrates device-driver interfaces in an example general-purpose system. The applications communicate through the OS to the necessary device drivers, which perform the hardware-specific communication and administration in order to fulfill the applications' I/O requests. With such a modular design of the device drivers, necessary changes to support different hardware is well-defined, simplifying the implementation of a given system.

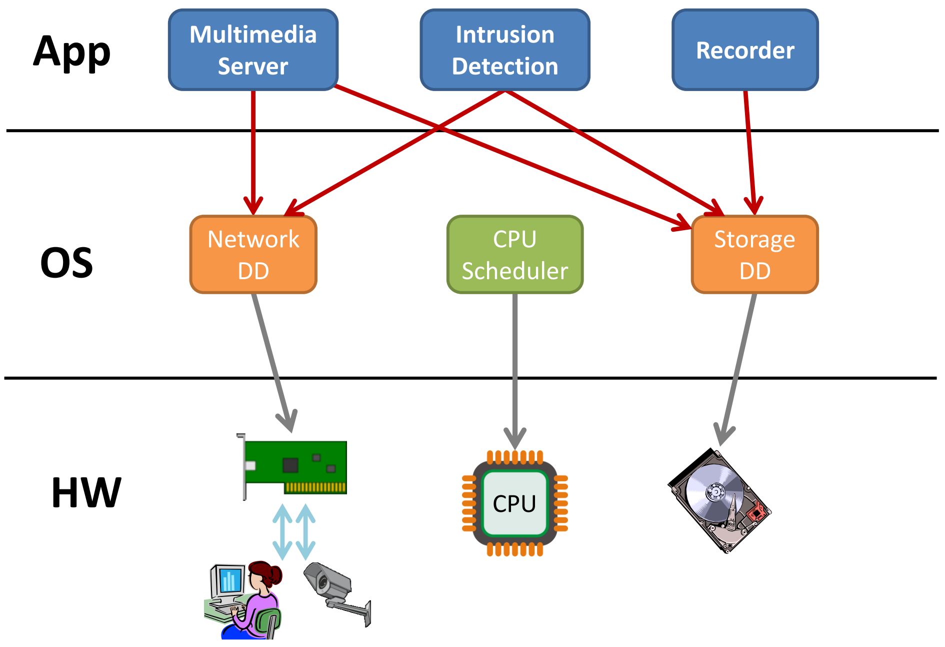


Figure 5.1: Devices driver context in a typical general-purpose system.

5.2.1 Allocation of Processor Time

Device drivers and supporting kernel code use the system's processor to execute software-encoded procedures for providing a given I/O service. The allocation of processor time for I/O service is expected to be provided in a timely manner but should not cause other time constrained applications also contending for processor time to miss their deadlines.

A logical approach to guarantee deadlines are met is to use existing real-time scheduling theoretical techniques by enumerate the workloads contending for processor time and create abstract workload models for them. The workloads on the system contending for processor time could be divided into the application and system software (e.g., device-drivers). For application software, one could write a computer program in such a way that it requires periodic allocations of processor time (e.g., 1msec every 10msecs). Such a program would be written using the OS's thread abstraction to periodically contend to execute a sequence of instructions that take no more than its determined WCET (e.g., 1msec). Upon completion of a single periodic execution the program would then suspend itself until the beginning of the next period. Assuming a fixed-task priority system, real-time scheduling theory would then be used to assign task priority values and determine whether or not deadlines may be missed with the given system parameters. A system model that is verified to meet all deadlines would then be run on the system by running the application program software as threads and assigning each with the scheduling theory assigned priority. Assuming the system and application are well-behaved (e.g., an application does not consume more than its WCET), no deadlines will be missed.

The approach used for above for designing application software cannot be easily applied to system software such as device drives. The problem is that most device driver and I/O framework software is not designed and implemented to be amenable to a real-time workload model. The device drivers are often written by 3rd parties (e.g., manufactures) with well-defined specifications for correct operation within a given GPOS. However, the particular execution of device drivers and associated OS code is often scheduled in an ad hoc manner through a combination of hardware invoked interrupt mechanisms and kernel scheduling heuristics that are designed to provide good average-case performance for the device's I/O. These scheduling heuristics for allocating processor time are generally not well-formed for straight-forward translation into real-time workload models, which is required for applying schedulability analysis. Further, mechanisms that are in-place for configuration applications do not always extend to device driver processor execution.

Assuming that a complete rewrite of the device driver and I/O framework is not an option, the problem is to determine an appropriate workload model them. For example, assume that one would like to model a given device driver's processor time as a periodic or sporadic task, what would be the period and WCET of such a workload model? Allocation of processor time for executing device driver code generally uses different mechanism from other application software, and therefore attempting to configure the device driver processor allocation into a given real-time workload models is often awkward. The end result is often that the abstract workload model is either invalid (breaking the assumptions of the theoretical analysis) due to lack of control over processor allocation or impractical/inefficient due to large WCETs in comparison average-case execution times. Further, many of the configurable application parameters (e.g., priority) do not extend to device driver processor allocations. That is, the explicit real-time constructs such as pre-emptive, priority-driven resource scheduling, real-time synchronization mechanisms, etc. are not typically available to easily configure device driver processor usage.

The processor usage of device drivers can often be substantial. Such processor usage may appear negligible for relatively slow devices such as hard disks and therefore not cause any problems due to various "fudge factors" (e.g., slight inflation of WCET) typically added to the system model to account for minor inaccuracies. That is, one might conclude that the speed differences between the processor and the hard disk mean that only small slivers of time will be consumed. However, the device driver overhead can be substantial for high bandwidth devices such as network cards. In fact, it is shown in Section 5.4, that processor usage for a Gigabit network device driver can be as high as 70%, which is large enough if not scheduled appropriately to cause missed deadlines.

The amount of processor time consumed by device drivers is not likely to diminish over time. Devices are becoming faster and utilizing more system resources. One example is the replacement of solid-state storage for hard disk drives. Solid-state devices are much faster and can significantly increase the density of processor interference experienced by other applications on the system over hard disk drivers. Further, the processor interference is not only limited to the device driver, but one can envisions such interference extending to the processor time required by a file system as well.

To better understand the problems with device drivers in the context of real-time scheduling, the following text will discuss the mechanisms by which device driver processor time is allocated. In particular, the discussion covers why it is difficult to configure processor allocation of I/O system software to correctly fit real-time abstract workload models with reasonable parameters.

Interrupts. Stewart [112, 113], lists improper use of interrupt handlers as a one of the top five most common pitfalls when developing embedded real-time software. Interrupt handlers often allow device drivers to obtain processor time regardless of the OS's scheduling policy. That is, while scheduling of application thread is carried out using the OS scheduler, the scheduling of interrupt handlers is accomplished through interrupt controllers typically implemented in hardware. Interrupts effectively create a hierarchy of schedulers, or two priority spaces, where all interrupts have a priority higher than other OS schedulable threads on the system. Therefore, the interrupt processor time results in priority inversion and must be considered in the abstract system model.

As an example of device driver code designed without real-time scheduling in mind, Jones and Saroiu [46] provided a study of a soft modem, which provides a real-world instance of a vendor produced device driver monopolizing a system's processor time through indiscriminate use of interrupt handlers. By performing signal processing required for the soft modem in interrupt context, the device driver is able to prevent other activities on a system from meeting deadlines.

Interrupts also may prevent even other interrupts handlers from running on the system until they have completed. While an interrupt is being handled, it is common on a GPOS for all interrupts to be disabled in order to ease the burden of correctly implementing mutual exclusion. The disabling of interrupts produces a blocking effect for other activities that may arrive on the system (e.g., arrival of a network packet). Until interrupts are re-enabled, no other threads can preempt the currently executing interrupt handler. Therefore, if a high-priority activity arrives while interrupts are disabled, this activity will have to wait until the interrupt completes, effectively reducing the time window a job has to complete its activities.

Interrupts can be disabled, which provides some coarse-grained control over interrupt invoked processor usage. For example, by disabling interrupts whenever a higher-priority task begins execution and re-enabling interrupts when no higher-priority tasks exists provides one possibility for interrupt priorities to be interleaved with the real-time task priorities. Unfortunately, enabling and disabling interrupts may have some unintended side effects. First, some devices require service from the CPU in a bounded amount of time. Without acknowledgment from the processor, the device may enter an unstable state and/or events may be lost. Other effects such as unnecessarily idling the device by not providing it with pending work, which can greatly reduce the utilization of certain devices. For instance, consider a hard disk drive. Once it completes a request, new requests, if any, are often desired to be passed immediately to the hard disk in order to allow the hard disk to start processing the request, likely increasing the overall throughput. If the hard disk interrupt handler is unable to execute due to interrupts being disabled, the disk may become idle, wasting time that could be used to service requests.

Disabling interrupts is not always a practical option. While commodity hardware generally provides a mechanism to disable particular interrupts (e.g., only disable the network card interrupt, while leaving the scheduler's timer interrupt enabled) [44], however, it is not always feasible to disable all interrupts. For instance, interrupts are typically used to invoke the scheduler in order to allocate processor real-time tasks. Therefore, disabling interrupts can have the unintended effect of causing *priority inversion* [57], a condition where a higher priority job is not executing and is not suspended, while at the same time a lower priority task is executing. In practice, priority inversion is common due to sections of code not allowing preemption known as a *critical section*. Preventing preemption is often necessary in order to allow access to shared data without the possibility of *race conditions*. Fitting critical sections into schedulability analysis has been addressed (e.g., [102]) by many researchers and is one consideration to fit interrupt handlers into schedulability analysis.

On some systems, enabling and disabling interrupts incurs significant CPU overhead, since manipulating the registers of the interrupt controller often involves off-chip access and may also cause effects such as pipeline flushes. Given that very few interrupts occur during the periods of masked interrupts, [114] proposed *optimistic interrupt protection*, which does not mask the interrupts using the hardware. To maintain critical sections, a flag is set that indicates a critical section is being entered and is cleared at the end of the critical section. If a hardware interrupt occurs during the critical section, an interrupt handler prologue will note that an interrupt has occurred, save the necessary system state, and update the hardware interrupt mask. The interrupted code will then continue. At the end of the critical section, a check will be performed for any deferred interrupts. If one does exist, the corresponding interrupt routine will then be executed. A similar approach can potentially be used for invoking the scheduler.

As evidence that general-purpose systems are not always designed with real-time implementing being the most important, general-purpose systems may contain interrupts that do not have any mechanism to disable known as *non-maskable interrupts* (NMIs). NMIs are often handled by the BIOS firmware rather than an OS installed handler. The most common form of NMIs are known as System Management Interrupts (SMI) and can result in additional latencies for real-time applications and therefore decrease the ability of a system to meet certain timing constraints [121].

Polling. I/O device events can often be detected by querying the device. Rather than allowing the device to control the allocation of processor time, another approach that is often used is to periodically query the device to determine if an event has occurred, which is referred to as *polling*. A discussion of the merits of interrupts and polling follows.

Interrupts allow the processor to perform various other work or to enter a low-power state without repeatedly using processor time to query the device. Further, with interrupt notification the time before detecting an event is generally, shorter than polling, since the interrupt notification delay is only due to the signal transmission from the device to the processor (assuming the interrupt is enabled), while the delay with polling can be as much as the largest interval between checks.

On the other hand, with polling, the processor communicates with the device at a time of the device-driver's choosing. This means that other activities on the system can continue undisturbed. For instance, the querying of the device can be scheduled to occur at a time that does not interfere with a higher-priority application. That is, the thread that queries the device can be under the direct control of the system's scheduler, thereby providing much more flexibility for scheduling notification of device events.

Interrupt execution can potentially consume a signification fraction of processor time. This phenomenon, known as *interrupt overload*, is pointed out by Regehr and Duongsaa [94]. Interrupt overload occurs when interrupts arrive at such a rate that the interrupt processing time used starves other activities on the system (including the OS scheduler). Several situations may cause an unexpectedly high interrupt rate. One is a faulty device continuously asserting interrupt signals, also known as a stuck interrupt. Another is simply a device that can send interrupts at high rates. In either case, servicing each interrupt handler upon the arrival of its corresponding interrupt can starve other activities on the system. In the extreme case, one could imagine that all processor time is spent executing interrupt handlers.

Network cards are an example of a device with a high maximum arrival rate. Often the network card can be configured to generate an interrupt upon arrival of a packet. Interrupts for packet arrivals may be desired in order to wake up and/or pass packets to threads waiting for information contained within the packet. A low response time for noticing this event is desired because there may be high-priority tasks awaiting the data, and delaying the delivery will increase the tasks' response time, possibly causing missed deadlines. Therefore, in many cases it is desirable to use interrupts on a system with real-time applications, since the I/O may be necessary for the real-time application to function correctly.

As long as interrupts are infrequent, and their handlers take a small amount of time, the impact on the system may be considered negligible. However, the increase in performance of the newer gigabit and higher Ethernet cards has the side effect of also increasing the maximum number of interrupt arrivals to a rate that can consume significant portions of CPU time. This means that the danger of interrupt overload is present on systems with these devices. For instance, the interrupt arrival rate of a gigabit Ethernet device can be nearly 1.5 million times per second [48, 94]. This arrival rate can overwhelm the processor with handling interrupts and leaves little or no time to perform other activities on the system.

To address the problem of interrupt overloads, [94] proposed to rate-limit interrupts via intelligently enabling and disabling interrupts. This mechanism will either delay interrupts or shed the interrupt load by dropping excessive interrupts to ensure that thread-level processing can make progress and not be blocked for extended periods of time due to a malfunctioning hardware device. The first approach enforces a minimum interarrival time between subsequent interrupts. The second approach caps the maximum number of interrupts in a given time interval. These solutions only count the number of interrupts arriving rather than calculating the actual amount of processor time the interrupts use, since counting incurs lower overhead. Also, for simple systems, the execution time of any given interrupt handler is nearly constant. On systems where the execution time of interrupt handlers can vary extensively or where the execution time depends on the state of the processor, counting interrupts alone may be insufficient.

In the context of network cards, [77] provides another solution to throttle interrupts. One key observation is that when the first receive interrupt arrives (signaling a packet has been received), the second receive interrupt is not useful until the first packet has been processed. Therefore, without completion of the first packet, the second interrupt and further interrupts are only informing the system of something it already knows, that is, the device needs attention. Therefore, [77] proposes to switch between interrupt and polling modes dynamically, so, once one interrupt arrives from the network device, interrupts are disabled. The work required to service the network device will then be performed outside the interrupt handler, typically in a schedulable thread. That thread will poll, querying the network device for work to perform once the previous unit of work has been completed. Once the network device has no further work to perform, the interrupts for the network device will be re-enabled and the receive thread will suspend itself, performing the transition back to interrupt mode and out of polling mode.

To provide some control over hardware interrupts, hardware platforms, such as the Motorola 68xxx series and some of the newer x86 microprocessors [45], provide the ability to set priorities for interrupt handlers within the same priority space as the system's schedulable threads. To provide this, the OS is able to set a priority value for each interrupt. The OS then sets a priority value for the currently running thread. When an interrupt arrives to the interrupt controller, if the interrupt priority value is greater than that of currently executing threads, then the processor will be interrupted. Otherwise, the interrupt processing must wait until the currently running thread's priority drops to a value below that of the interrupt. Assigning priorities for interrupts does alleviate some of the problems with interrupts. However, this mechanism is not available on all hardware platforms and does require OS support.

Processor Allocation as Threads. Processing interrupts at an exclusively higher priority than the OS's schedulable threads tends to result in a less configurable system. That is, it is often desired to configure I/O service at a priority higher than some threads but lower than others. One approach is to force interrupt handling into OS schedulable threads and thereby use the same set of priority values as other applications on the system. Providing a means to assign priority values to all processor allocations, provides more flexibility and therefore seemingly allowing the system to be configured to more closely match ideal theoretical scheduling models.

Moving processor allocation from interrupt handlers to OS schedulable threads was first proposed in the paper "Interrupts as Threads" [52]. The paper was motivated by unnecessary delays experienced when disabling all interrupts in order to achieve mutual exclusion between OS schedulable threads and interrupt handlers. In particular, interrupts handlers that do not require mutual exclusion are unnecessarily delayed due to the coarseness of disabling all interrupts. Therefore, by converting interrupt handlers into schedulable threads, only those processor allocations that would violate the mutual exclusion experience delays.

The technique outlined in [52] to convert interrupt handlers into thread is described as follows. When an interrupt arrives, the interrupted thread's state is changed to non-runnable. Changing the state is needed because the interrupt uses the interrupted thread's stack. Next, the interrupt handler is moved to one of a set of pre-allocated thread stacks. Once the interrupt thread is migrated to its own stack, it begins execution on the processor. At this point, it is important to realize the interrupt handler has not yet been made into a full-fledged thread, which requires more operations such as flushing the register windows from the interrupted thread. The interrupt handler is only made into a full-fledged thread if the handler should logically be blocked in order to maintain mutual exclusion, thereby reducing the overhead of "threading" an interrupt handler. Once the handler completes or blocks, the interrupted thread is set as runnable and the scheduler chooses the next thread to execute. Allowing interrupts to use the same synchronization primitives as OS schedulable threads means that the interrupts no longer have to be disabled in order to achieve synchronization. This results in decreased response time for activities triggered from interrupts. Experimental results illustrated that their technique reduced response time from over a second to well under a millisecond.

By forcing all interrupt handlers to be implemented as OS schedulable threads increases the flexibility to configure a given system and therefore generally increases the number and/or type of time constrained applications that can be supported. The approach of threading all interrupt handlers is taken by the RT PREEMPT version of the Linux kernel [28, 34, 74]. The way interrupt handlers are converted to OS schedulable threads is illustrated in Figure 5.2 and described as follows. In both the RT and vanilla versions of the kernel, a generic OS interrupt handler is installed for all interrupts on the system. However, on the RT version of the Linux kernel, the response by the kernel to a request by a device driver to install an interrupt handler is to create a new OS schedulable thread that executes the requested device interrupt handler upon being unblocked. Therefore, when a given hardware component raises an interrupt the generic interrupt handler changes the device driver's interrupt handler thread to runnable. Further, it should be noted that the interrupt is also not acknowledged as is typical in interrupt handlers and therefore, the maximum amount of unavoidable interference by a given device is limited to one execution of the generic OS interrupt handler. The final action of the generic OS interrupt handler is to run the scheduler code to determine whether if a different thread (from the interrupted thread) should be allocated the processor.

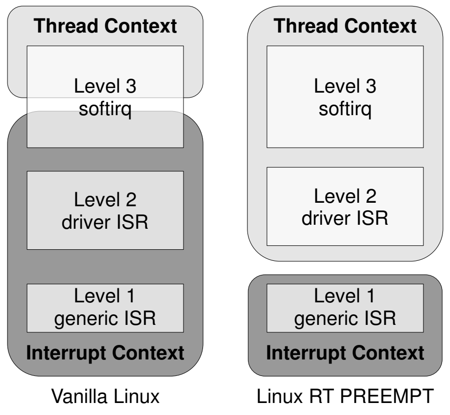


Figure 5.2: Threading of interrupt handlers in the RT PREEMPT version of the Linux kernel.

5.3 Demand-Based Schedulability Analysis

The following discussion reasons about meeting deadlines through a demand-based analysis approach using existing studies on the traditional workload models of a *job* and *task* [6, 9, 12, 67] as discussed in Chapter 3.

The *demand* of a job is the cumulative amount of processor time allocated to it (or it consumes) over a given time interval. Using notation, the demand of job J over the time interval $[a, b)$[1] is $demand_J(a, b)$.

The amount of interference experienced by a job will be used to calculate whether a job is allocated a sufficient amount of time at or before its deadline. Intuitively, the interference of a job is the amount of time the processor is allocated to another job(s) between the job's release time and the time instant it completes. Interference can be thought of as the inability of a job to obtain processor time, thereby delaying the timely completion of a job. A job's completion time is the time

[1]Brackets ([]) and parentheses are used to signify whether or not the endpoints of a given time interval are included. As an example, $[t_1, t_2)$ represents the time interval consisting of all time instants t such that $\{t \mid t_1 \leq t < t_2\}$. Using the same notation, the length of a time interval is defined to be $t_2 - t_1$, regardless of whether or not the endpoints are included.

instant it has been allocated a cumulative amount of processor time equal to its execution time. A job is said to meet its deadline if its completion time is $\leq$ its deadline.

Assuming a single processor and a work-conserving scheduling algorithm,[2] the total interference a job experiences can be computed by summing the total amount of higher-priority demand in the job's demand window. A job J_k released at time instant r_k and completing at time instant c_k has a demand window equal to $[r_k, c_k)$. In words, the demand window of a job is the time interval over which it is competing for processor time. Therefore, the amount of interference a job experiences can be computed by summing the total amount of higher priority demand. Therefore, a job completes by its deadline if the sum of its own execution time plus the total amount of interference is less than or equal to its deadline. Consider a set of jobs $\mathcal{J} = \{J_1, J_2, ...\}$. Let the priority of a job J be its subscript. Further, let a lesser number indicate a higher priority (e.g., J_1 has a higher priority than J_2). A job J_k will meet its deadline if the following is true:

$$e_k + \sum_{i<k} demand_{J_i}(r_k, r_k + d_k) \leq d_k \tag{5.3.0.1}$$

As an illustration, consider Figure 5.3. If the maximum interference plus the WCET of a given job does not exceed the length of the time interval from a job's release time to its deadline, then the job must be allocated sufficient processor time for it to complete before its deadlines.

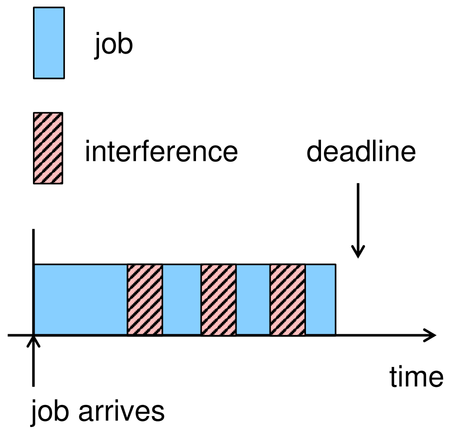


Figure 5.3: Sufficient processor time available for a job to complete before its deadline.

[2]This discussion (and more generally this entire dissertation unless stated otherwise) assumes that the scheduling algorithm will not intentionally idle processor (or I/O resource), a property commonly referred to as *work-conserving*. This property simplifies the analysis since the scheduling algorithm is assumed to not contribute to a job's interference.

The same reasoning for verifying that jobs meet their deadlines can be extended to periodic and sporadic tasks by calculating an upper-bound on the amount of interference any job in a task experiences. The maximum demand function of a task, $demand^{\max}_{\tau_i}(\Delta)$, is the cumulative amount of demand of all jobs of τ_i in any time interval of length Δ. Assuming that a task represents a number of possible sequences of jobs where a given sequence is S and all possible job sequences is $\mathcal{S}$, the maximum demand of a task considers all such sequences over all time intervals of length Δ as follows:

$$demand^{\max}_{\tau_i}(\Delta) \stackrel{\text{def}}{=} \max_{S \in \mathcal{S}, t>0} \sum_{J \in S} demand_J(t - \Delta, t) \tag{5.3.0.2}$$

Using the above calculation for interference of tasks, all jobs of a task τ_k will complete by their deadline if:

$$e_k + \sum_{i=1}^{k-1} demand^{\max}_{\tau_i}(d_k) \leq d_k \tag{5.3.0.3}$$

The periodic constraint of periodic and sporadic tasks limits the amount of demand a given task can generate in a time interval, since the release of two subsequent jobs must be separated by at least a period length of time. That is, only one job, starting at an arbitrary time instant t, can be released in the time interval $[t, t + p_i)$, two in $[t, t + 2p_i)$, ..., an upper-bound on the amount of demand generated by task τ_i in any time interval of size $\Delta > 0$ is:

$$demand^{\max}_{\tau_i}(\Delta) \leq \left\lceil \frac{\Delta}{p_i} \right\rceil e_i \qquad \text{if } \Delta > 0 \tag{5.3.0.4}$$

The research of this dissertation introduces a *refined demand bound.* The refinement is based on the observation that the amount of demand (i.e., processor time) a job can be allocated over a given time interval cannot exceed the length of the time interval. The previous analysis assumes that the interference experienced by a job is the sum of the total higher priority demand. The higher priority demand is calculated by summing each higher priority job's execution time immediately at each release time instant. However, calculating the interference in such a way is pessimistic since a job cannot be allocated an amount of processor time greater than the size of the considered time interval. For example, consider a task τ_i with $p_i = d_i = 7$, $e_i = 2$, and $\Delta = 8$. The calculated upper-bound using Equation 5.3.0.4 is 4. However, the release times of any two subsequent jobs of

τ_i must be separated by at least 7 time units (periodic constraint) and therefore a tighter upper bound would be 3 time units. This upper bound on the demand of 3 time units can be clearly seen in Figure 5.4. Regardless of where the Δ-sized interval is positioned in time, the maximum execution time cannot be larger than 3 time units.

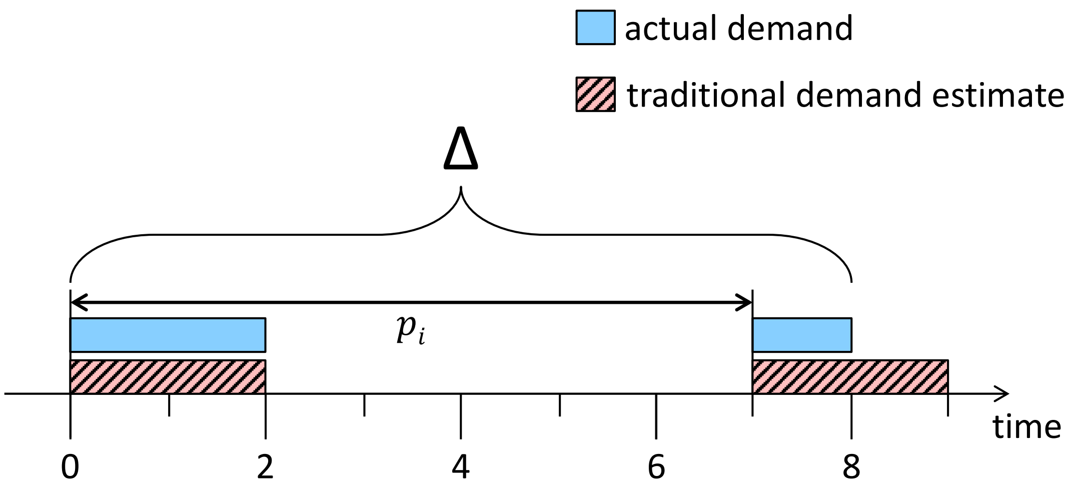


Figure 5.4: Comparison of the traditional demand bound calculation with the actual demand of a periodic task τ_i ($p_i = d_i = 7$ and $e_i = 2$).

The refined demand bound of a task, τ_i can be calculated on a time interval of length Δ by considering the execution demand of all full-periods plus the demand of the final time interval as follows:

$$demand_{\tau_i}^{\max}(\Delta) \leq je_i + \min(e_i, \Delta - jp_i) \tag{5.3.0.5}$$

where

$$j \overset{\text{def}}{=} \left\lfloor \frac{\Delta}{p_i} \right\rfloor$$

The first term on the right hand side of the inequality is the demand for all full periods (j can be thought of as the number of jobs for all full periods). The second term is the maximum amount of execution time in the final, not yet completed, period-sized time interval. Again, the min is due to a job only being able to consume an amount processor time equal to the amount of time after its release and no more than e_i.

An illustration of the refined and traditional demand bound functions for τ_i with $p_i = d_i = 7$, $e_i = 2$, and $\Delta = 8$ is illustrated in Figure 5.5.

A less refined, but simpler demand bound, can be calculated by interpolating linearly between the e_i length of time after each job's release time. That is, $\Delta = jp_i + e_i$ for each $j = 1, 2, 3, \cdots$, the points where the traditional and refined demand bound converge. The expression on the right-hand of the inequality in Equation 5.3.0.5 for these points is:

$$(\frac{\Delta - e_i}{p_i} + 1)e_i = u_i(\Delta + p_i - e_i)$$

resulting in a piecewise linear function for $\Delta > 0$

$$demand^{\max}_{\tau_i}(\Delta) \leq \min(\Delta, u_i(\Delta + p_i - e_i)) \tag{5.3.0.6}$$

A comparison of the three demand bound functions is illustrated in Figure 5.5.

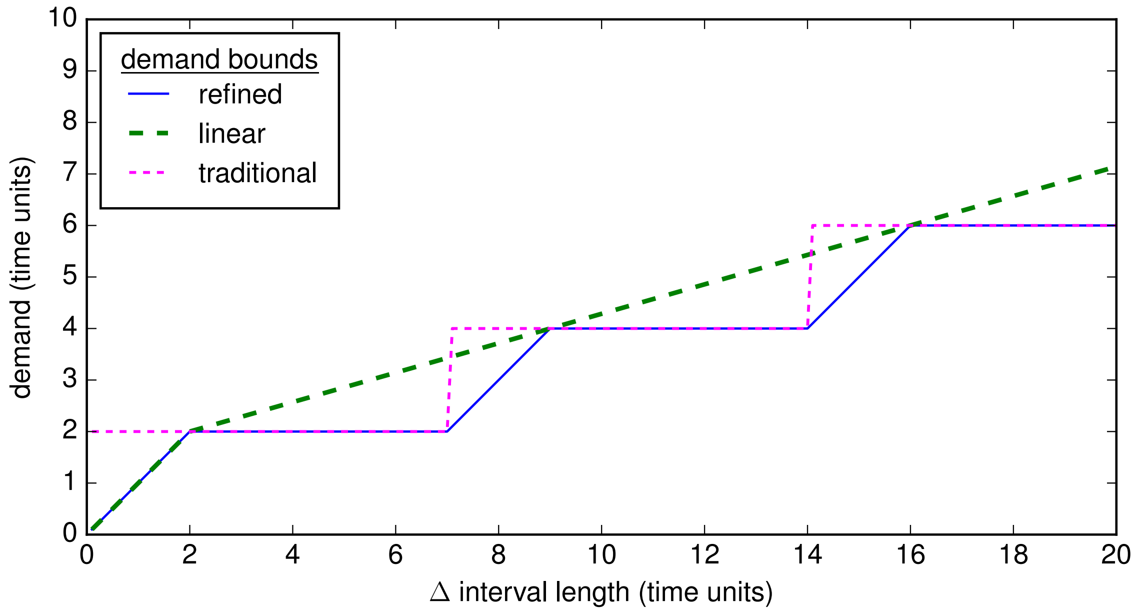


Figure 5.5: Comparison of the demand bound calculations for a periodic task with $p_i = d_i = 7$ and $e_i = 2$.

An arguably more intuitive formulation of processor demand is achieved by expressing the ratio of demand to interval length, which for this dissertation is called *load*. More precisely, $load_{\tau_i}(t-\Delta, t) \overset{\text{def}}{=} demand_{\tau_i}(t - \Delta, t)/\Delta$ and $load^{\max}_{\tau_i}(\Delta) \overset{\text{def}}{=} demand^{\max}_{\tau_i}(\Delta)/\Delta$. It follows from (5.3.0.3) that a task τ_k will always complete by its deadline if:

$$\frac{e_k}{d_k} + \sum_{i=1}^{k-1} load^{\max}_{\tau_i}(d_k) \leq 1 \tag{5.3.0.7}$$

In words, the above equation verifies that all deadlines of a task τ_k are met by summing the maximum percentage of processor time for all higher priority tasks and itself over a time interval equal to its deadline d_k. If the sum does not exceed 100%, then e_i time units will always be allocated to any given job in τ_k at or prior to its deadline.

Similarly, a *refined load bound* can be formulated by dividing Equation 5.3.0.5 by Δ:

$$load^{\max}_{\tau_i}(\Delta) \leq \frac{je_i + \min(e_i, \Delta - jp_i)}{\Delta} \tag{5.3.0.8}$$

where j is defined as in (5.3.0.5).

Additionally, a simplified *hyperbolic load bound* can be obtained by dividing (5.3.0.6) by Δ as follows:

$$load^{\max}_{\tau_i}(\Delta) \leq \min(1, u_i(1 + \frac{p_i - e_i}{\Delta})) \tag{5.3.0.9}$$

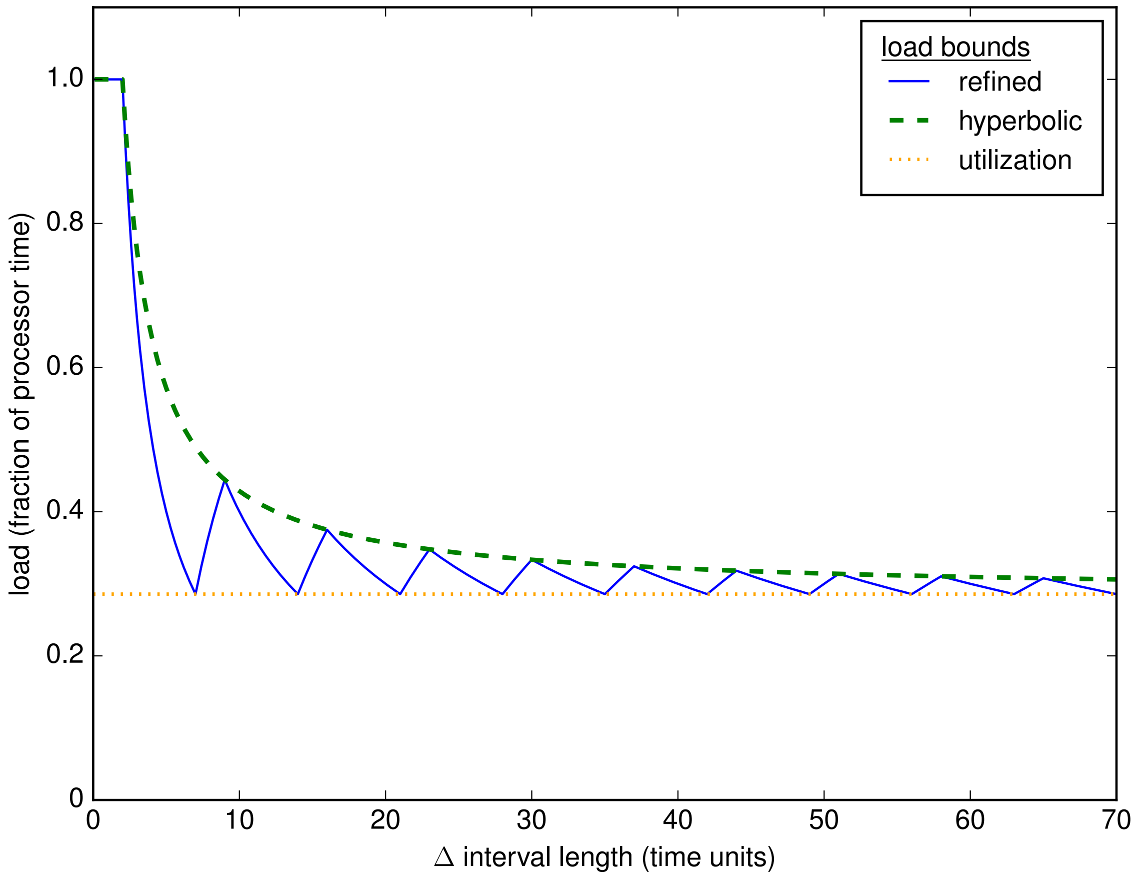


Figure 5.6: Plot of the theoretical load bounds for a periodic task with $p_i = 7$ and $e_i = d_i = 2$.

These above load bounds will provide the basis for the empirical approach to model I/O service processor time and is presented in the following section.

5.4 Technique to Derive a Load Bound Model from Empirical Measurements

This section presents an empirical measurement-based technique to derive an I/O service's load-bound function, which can then be used in schedulability analyses as described in the previous Section 5.3. The implementation of I/O services (e.g., IP network packets) in a general-purpose OS is not designed to consume processor time in such a way to conform to a periodic or sporadic task model. Given that a workload model is not available, the results obtained by applying load-based analysis will not be correct due to I/O service induced processor interference not be considered. Therefore, the purpose of this section is to formulate a workload model that accurately represents the consumption by the system's I/O service and can be used in the load-based schedulability analysis. The approach is to use a set of empirical measurements of the consumed processor time by the I/O service in order to derive a sporadic task model, which can then be used with to perform theoretical load-based schedulability analysis.

The initial intuition when developing the following measurement technique is that the I/O service processor time can be viewed as a conceptual sporadic task and therefore can be characterized with a load-bound function and corresponding plot similar to Figure 5.6. Suppose one wants estimate of the worst-case I/O service processor load. Such a load can be viewed as a conceptual task, which we will refer to as τ_D. Performing load-based schedulability analysis requires the τ_D's corresponding load-bound function, $load^{\max}_{\tau_D}(\Delta)$. The value of $load^{\max}_{\tau_D}(\Delta)$ for various Δ-sized intervals is approximated by the maximum *observed* (i.e., measured) value of $demand_{\tau_D}(t-\Delta, t)/\Delta$ over a large number of intervals $[t-\Delta, t)$.

5.4.1 Measuring Processor Time

One possible approach to measure the processor demand of τ_D in a given-sized time interval is to modify the kernel, including the softirq and interrupt handlers, to track and record every time interval during which the τ_D executes. Initially, this dissertation research started with such an approach, but concerns arose from the complexity and the additional overhead introduced by including a fine-grained time accounting framework. Instead, a subtractive approach was used, in which the processor demand of I/O service is inferred by measuring the processor time that is left for other tasks. The following text describes this subtractive approach.

Previous research by Regehr [93, 95] discussed a technique to measure the amount of processor time that a given application thread did not receive, even though the application thread was configured with the highest priority on the system. Regehr pointed out that device drivers can in effect "steal" processor time from real-time applications and therefore result in them missing deadlines. In order to illustrate and quantify this stolen time, Regehr describes a technique for an application-level thread to monitor its own execution time without special OS support. The implementation of such a measurement program is called *Hourglass*. Hourglass uses an application thread, which we call *an hourglass thread*, to monitor the amount of processor time it is able to consume over a given time interval. The design of the hourglass thread is to provide a cross-platform technique that is able to measure the amount of processor time a given thread is allocated. That is, the technique should be easily portable to other OSs and therefore not rely on OS-specific internal instrumentation. How to measuring the amount of processor time received with such constraints is not immediately obvious because processor allocation is typically broken up into many small time slices, interrupt processing, awakened threads, etc., and the endpoints of these intervals of execution are not explicitly conveyed to a given thread. In order to overcome such the lack of explicit processor allocation information, an hourglass thread infers the times of its transitions between executing and not executing, by reading a fine-grained clock in a tight loop. If the time between two successive clock values is small, it assumes that no preemption has occurred. However, if the difference is large, then the thread is assumed to be preempted. Using this technique to determine preemption points, an hourglass thread can deduce the start and stop times of each execution interval, and calculate the amount of processor time it receives in that interval. Knowing the amount of execution time provides a means to simulate a real-time task. For example, a periodic workload can be emulated by having an hourglass thread alternate between states of contention for the processor and self-suspension. More specifically, a periodic hourglass thread contends for the processor (executes the clock-reading loop) until it receives its nominal WCET, and then suspends itself until the beginning of its next period. Further, an hourglass thread can observe whether or not it is has been allocated its WCET before its deadline. By adapting the hourglass thread to measure and record the *maximum* amount of "stolen time" an estimated load bound function can be formulated.

An estimate of a given theoretical, load-bound function can be derived by extending Regehr's Hourglass technique. The fraction of processor time value for a given time interval length, Δ, on a load bound function can be measured in an actual system by measuring the amount of time a task is preempted (not executing) over a time interval of length Δ in which the maximum amount preemption is experienced. By measuring a sufficient number of Δ-sized time intervals, one can derive an estimate of a theoretical load-bound function.

While encountering (and measuring) a time interval with the maximum amount of preemption may be very unlikely, one can create conditions that make it more likely to encounter a time interval with an amount of preemption near to the actual maximum. First, measurements should be taken while the cause of preemptions is causing a large amount of interference (e.g., by invoking a task's worst-case). Next, a large number of samples should be taken. That is, the assumption is that the larger the number of samples the more likely it is to encounter a time interval at or near the maximum.

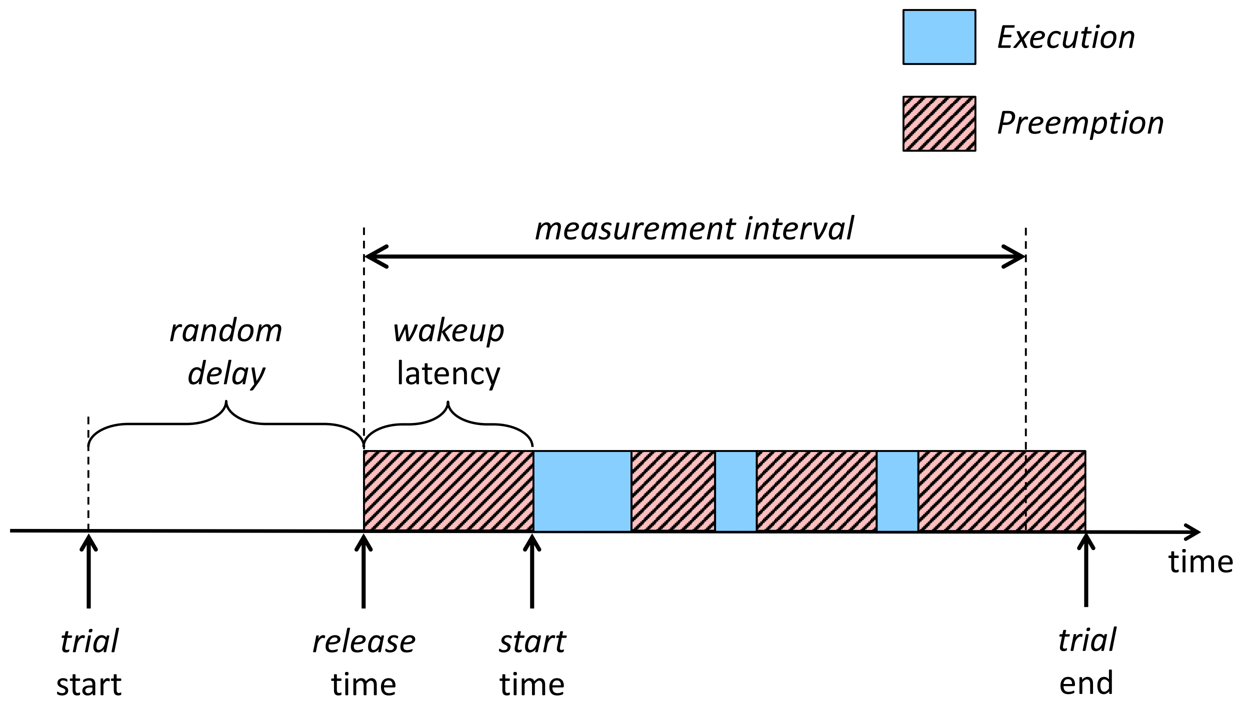


Figure 5.7: Technique to empirically deriving load bound functions from execution time measurements.

The amount of preemption in a given-sized time interval can be measured by programming an hourglass-like thread to contend for processor time over the desired time-interval. The thread then measures the amount of time consumed. Finally, to compute the amount of preemption one would simple subtract the measured execution time from the size of the time interval. Figure 5.7 illustrates one measurement instance (trial) of a given-sized time interval. The random delay is used to remove the effect of phasing between activities on the system from the measured values. Using this measurement technique, an estimate of the load bound graph is found by recording the maximum preemption experienced over all trials of each time interval considered.

A preliminary experiment to demonstrate the applicability of using empirical measurements to derive a load bound was performed by measuring a simple periodic task. The experiment consisted of a periodic task, implemented using the Hourglass technique, and a measurement task as described above. The results of the experiment are shown in Figure 5.8.

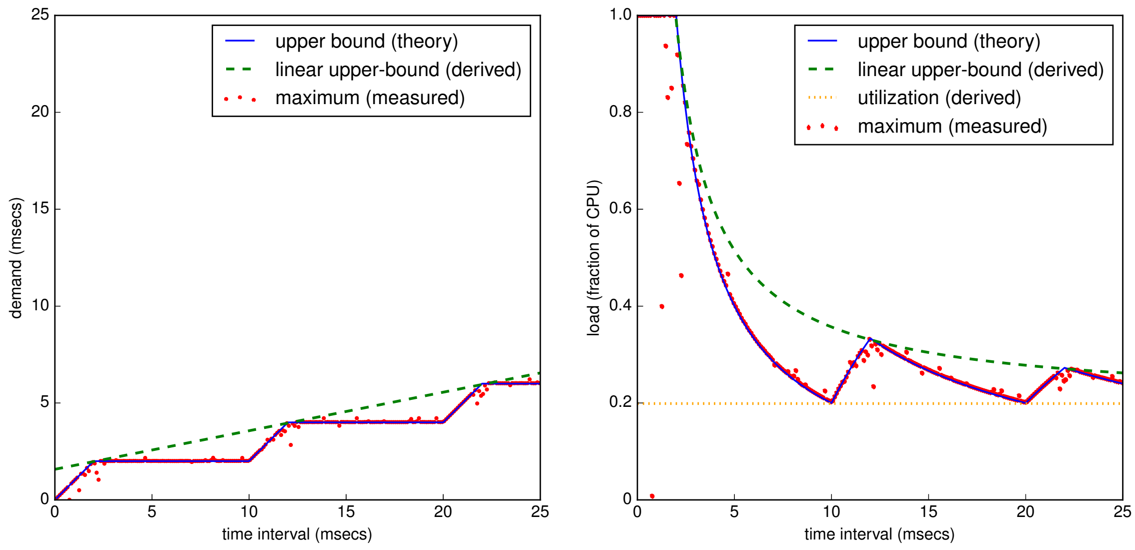


Figure 5.8: Measured demand and load bounds of a periodic task with a WCET=2 msecs and period=10 msecs.

The upper bounds marked derived in Figure 5.8 are derived from the measurement data as follows. (1) for each data point, if the maximum demand (y-value) of all data points with a smaller interval length (x-value) is found the data point is replaced by this maximum. (2) a linear lower bound on demand is formulated by finding the line that minimizes the summed vertical distance from the line to each data point. Each step is described in more detail below.

(1) A given demand measurement can be replaced by the maximum of all measured values with a smaller interval length for deriving a tighter lower-bound on the actual maximum demand. That is, given a demand measurement $demand^{msr}_{max}(\Delta)$ over some time interval of length Δ, the actual maximum demand, $demand^{actual}_{max}(\Delta)$ must be at least as large as all demand measurements of smaller time intervals (e.g., data points left of a given point in Figure 5.8).

Theorem 5.4.1
If the following conditions are true:

- $\Delta_1 < \Delta_2$
- $demand^{msr}_{max}(\Delta_1) \geq demand^{msr}_{max}(\Delta_2)$

then

$$demand^{msr}_{max}(\Delta_1) \leq demand^{actual}_{max}(\Delta_2).$$

Proof*(by contradiction).*
Suppose

$$demand^{msr}_{max}(\Delta_1) > demand^{actual}_{max}(\Delta_2) \tag{5.4.1.1}$$

By definition, the maximum demand over any length of time of size Δ must be at least the amount in an arbitrary time interval that is of length Δ. Therefore, $demand^{actual}_{max}$ is monotonically increasing. Further,

$$demand^{msr}_{max}(\Delta) \leq demand^{actual}_{max}(\Delta) \tag{5.4.1.2}$$

It follows then that,

$$demand^{actual}_{max}(\Delta_1) > demand^{actual}_{max}(\Delta_2) \tag{5.4.1.3}$$

which is a contradiction since $demand^{actual}_{max}$ is a monotonically increasing function. □

(2) The set of all lines with slope ≥ 0 is found that both; (a) touch at least one data points and (b) upper-bound all other measured points. For each line, the summed vertical distance from the line to each measured point is calculated and the minimum is chosen as the linear lower bound on demand. From the linear demand bound, the function for the lower bound on load is derived by taking the linear demand bound and dividing by the time interval.

While the empirical measurements closely match the shape of the theoretical task, the majority of the measurements are slightly greater than the theoretical predicted maximum demand, which should not occur since the measurements are assumed to be a lower bound on the maximum demand of the periodic task being measured. However, one important aspect that is easily overlooked is that the OS itself consumes processor time and may contribute to the measurement task's interference. For instance, the kernel's processing of a periodic timer interrupt used to invoke updating of the kernel's state (e.g., current time). Such activities often execute as part of an interrupt handler, which is effectively a higher priority than all other user-space threads (e.g., the measurement thread). Further, non-preemptible sections also may contribute to the measured interference (e.g., sections implemented by disabling interrupts).

The inclusion of high priority and non-preemptible OS activities is critical to obtaining valid results from theoretical schedulability analysis. Generally, the assumption of theoretical schedulability analysis is that workload model, in this case a periodic task model, characterizes the worst-case behavior of a given real-world activity. Therefore, given set of periodic tasks the theoretical analysis can determine whether deadlines will ever be missed. However, if other activities such as the OS activities are not included in the analysis, one can easily see that such a discrepancy can easily result in real-world missed deadlines.

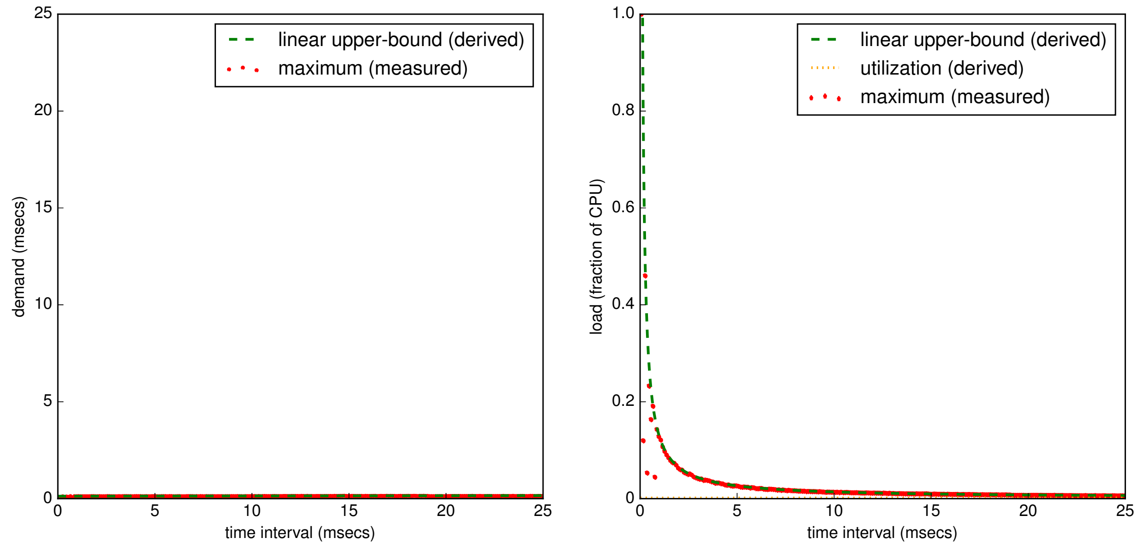


Figure 5.9: Interference measurements on an idle system.

In order to model the system activity and non-preemptible sections, the measurement task was run on an "idle" system and Figure 5.9 shows the resulting data. The idle system load can then be summed with the theoretical load of a periodic task to provide a more accurate upper-bound as shown in Figure 5.8. As seen in the figure, the resulting sum provides a valid upper-bound for the measured points (all measured points are less than or equal to the summed bound).

The load bound of a system's network service can be measured to derive a load bound function in a similar way as the periodic and system load. In our measured system, the load an IP over Ethernet-based network service is studied. The experimental setup consists of two computers, referred to as Host A and Host B, connected to a dedicated network switch. Host A sends network packets to host B at a rate to invoke the maximum processor demand of B's network service. On B, an application thread, τ_2, is assigned a priority lower than that of the network I/O service and continuously attempts to consume processor time. All other activities on C are either shut down or run at a priority lower than that of τ_2. If Δ is the length of the time interval currently being measured, τ_2 continuously chooses Δ-sized time intervals and measures the amount of processor time it is able to consume in them. So, if τ_2 is able to consume x units of processor time in a given Δ-sized interval being measured, the processor demand attributed to the network I/O service is concluded to be $\Delta - x$ and the corresponding load is $(\Delta - x)/\Delta$.

It is important to note that the measurement thread is likely to only measures processor interference and not memory bus interference. The set of processor instructions and data that the measurement thread uses will often be relatively small compared to the size of the processor's cache and therefore most, if not all, of the measurement thread's memory accesses are likely to be served by processor's cache and therefore do not contend for access to the memory bus. Such an effect, commonly called *cycle stealing*, can slow down a memory-intensive task, but is considered to be outside the scope of this dissertation.

The following experiments used two machines, which correspond to Host A and B, each with a Pentium D dual-core processor configured to use only a single core, 2 GB of main memory, and an Intel Pro/1000 gigabit Ethernet adapter. The two machines were each connected to a dedicated gigabit network switch. The vanilla and the Timesys patched versions of the Linux 2.6.16 were installed on the machines.

Task τ_2 (i.e., measurement thread) was set to the SCHED_FIFO scheduling policy (fixed-task preemptive priority scheduling with FIFO service among threads of equal priority) and a real-time priority value just below that of the network I/O service softirq threads. All of τ_2's memory was locked into physical memory so other activities such as paging and swapping would not be measured.

The first experiment provides base-line measurements from an idle system. In this case no network packets were sent to the measurement machine. This measures the amount of load τ_2 can place on the system with no network I/O service processor load. The load value of τ_2 is then subtracted from 1 to obtain the idle system load. This provides a basis for isolating the network I/O service load by subtracting the idle interference from measured network interference.

The results of the idle-network measurement are shown in Figure 5.10, which is the amount of interference percentage observed by τ_2. Each data point of this and subsequent plots represents the maximum observed interference over a number of same-sized time intervals. Note that the plotted measurement is a hard lower bound and a statistical estimate of the given time interval being measured. Assuming that the interference and the choice of trial time-interval lengths are independent, the observed maximum should converge to the system's actual worst-case interference.

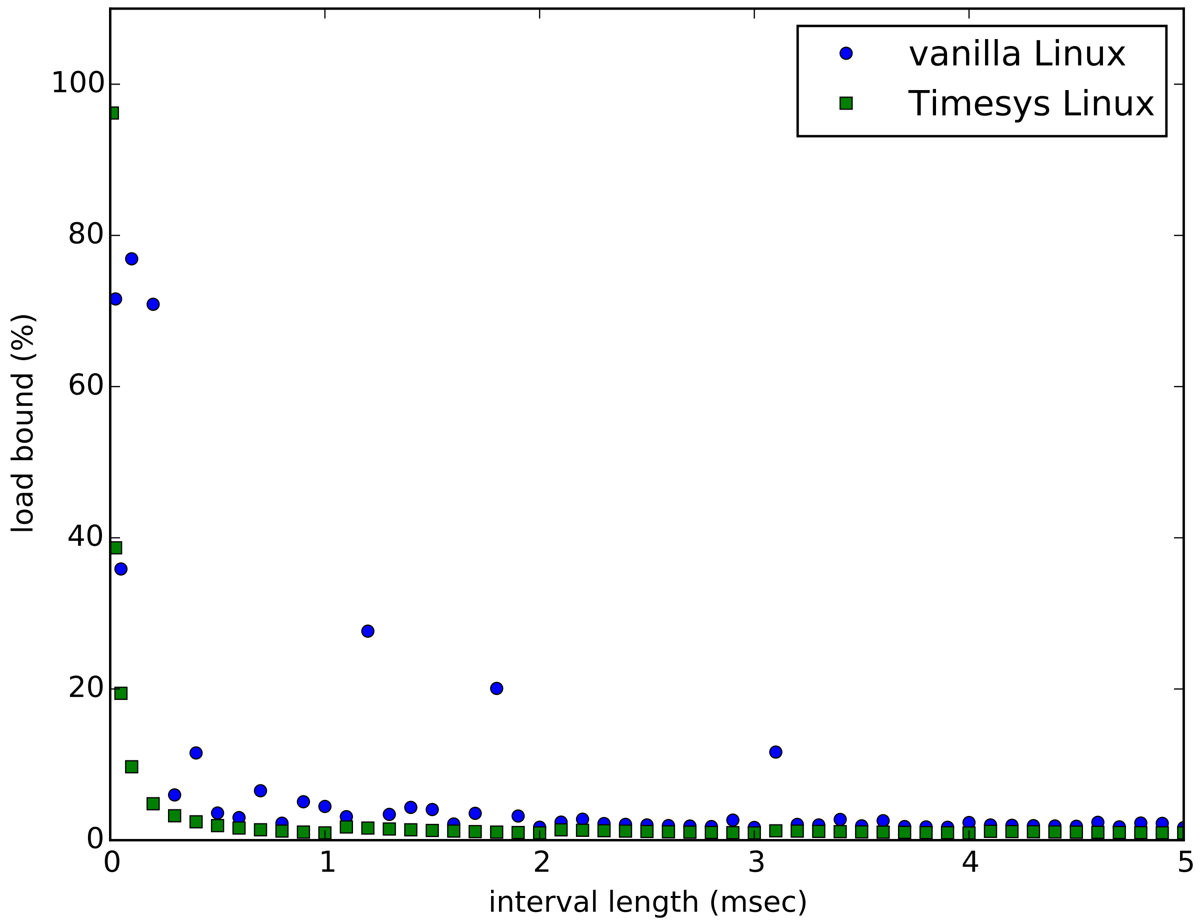


Figure 5.10: Observed interference without network traffic (i.e., idle network thread).

The measured data points should be approximately hyperbolic. By measuring a small enough time interval the maximum interference would be 100% and with larger time intervals the interference should converge to the utilization of the interfering task(s). Two likely reasons that the measured data points deviate from a hyperbola are: (1) the demand is periodic or nearly periodic resulting in the saw-tooth shape of the *refined load bound* (e.g., Figure 5.6); (2) the worst-case demand was not observed due to insufficient sampling. The reason for the latter deviation should be diminished by increasing the number of samples, while the former should not change.

In the case of Figure 5.10 the tiny blips for in the Timesys data around 1 and 2 msec are likely due to processing for the 1 msec timer interrupt. The data points for vanilla Linux exhibit a different pattern, aligning along what appear to be multiple hyperbolas. In particular, there is a set of high points that seems to form one hyperbola, a layer of low points that closely follows the Timesys plot, and perhaps a middle layer of points that seems to fall on a third hyperbola. This appearance is what one would expect if there were some rare events (or co-occurrences of events) that caused preemption for long blocks of time. When one of those occurs it logically should contribute to the maximum load for a range of interval lengths, up to the length of the corresponding block of preemption, but it only shows up in the one data point for the length of the trial interval where it was observed. The three levels of hyperbole in the vanilla Linux graph suggest that there are some events or combinations of events that occur too rarely to show up in all the data points, but that if the experiment were continued long enough data points on the upper hyperbola would be encountered for all interval lengths.

Clearly the vanilla kernel is not as deterministic as the Timesys kernel. The high variability of data points with the vanilla kernel suggests that the true worst-case interference is much higher than the envelope suggested by the data. That is, if more trials were performed for each data point then higher levels of interference would be expected to occur throughout. By comparison, the observed maximum interference for Timesys appears to be bounded within a tight envelope over all interval lengths. The difference is attributed to Timesys' patches to increase preemptibility.

The remaining experiments measured the behavior of the network device driver task τ_D under a heavy load, consisting of ICMP "ping" packets sent to the experimental machine every 10 μsec. ICMP "ping" packets were chosen because they execute entirely in the context of the device driver's receive thread, from actually receiving the packet through sending a reply (TCP and UDP split execution between send and receive threads). An illustration of the experimental setup is shown in Figure 5.11.

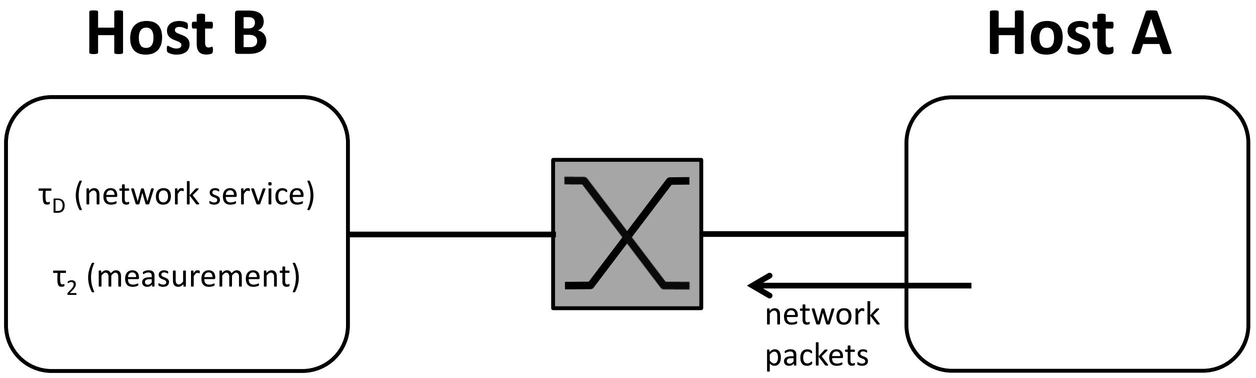


Figure 5.11: Host A invoking network processor interference on Host B by sending network packets at a high rate.

Figure 5.12 shows the observed combined interference of the driver and base operating system under a network load of one ping packet every 10 μsec. The high variance of data points observed for the vanilla kernel appears to extend to Timesys as well. This indicates a rarely occurring event or combination of events that occurs in connection with network processing and causes long periods of preemption. This may be a "batching" effect arising from the NAPI policy, which alternates between polling and interrupt-triggered execution of the driver. An observation taken from the data is that the worst-case preemptive interference due to the network driver is higher with the Timesys kernel than the vanilla kernel. A likely explanation for the higher interference is the additional time spent in scheduling and context-switching since the network softirq handlers are executed in OS scheduled thread context rather than a borrowed context (e.g., interrupt handler).

Given a set of data from experimental interference measurements, a hyperbolic bound can be fit through application of inequality (5.3.0.8) in Section 5.3. There are several ways to choose the utilization and period so that the hyperbolic bound is tight. The method used here is: (1) eliminate any upward jogs from the data by replacing each data value by the maximum of the values to the right of it, resulting in a downward staircase function; (2) approximate the utilization by the value at the right most step; (3) choose the smallest period for which the resulting hyperbola intersects at least one of the data points and is above all the rest.

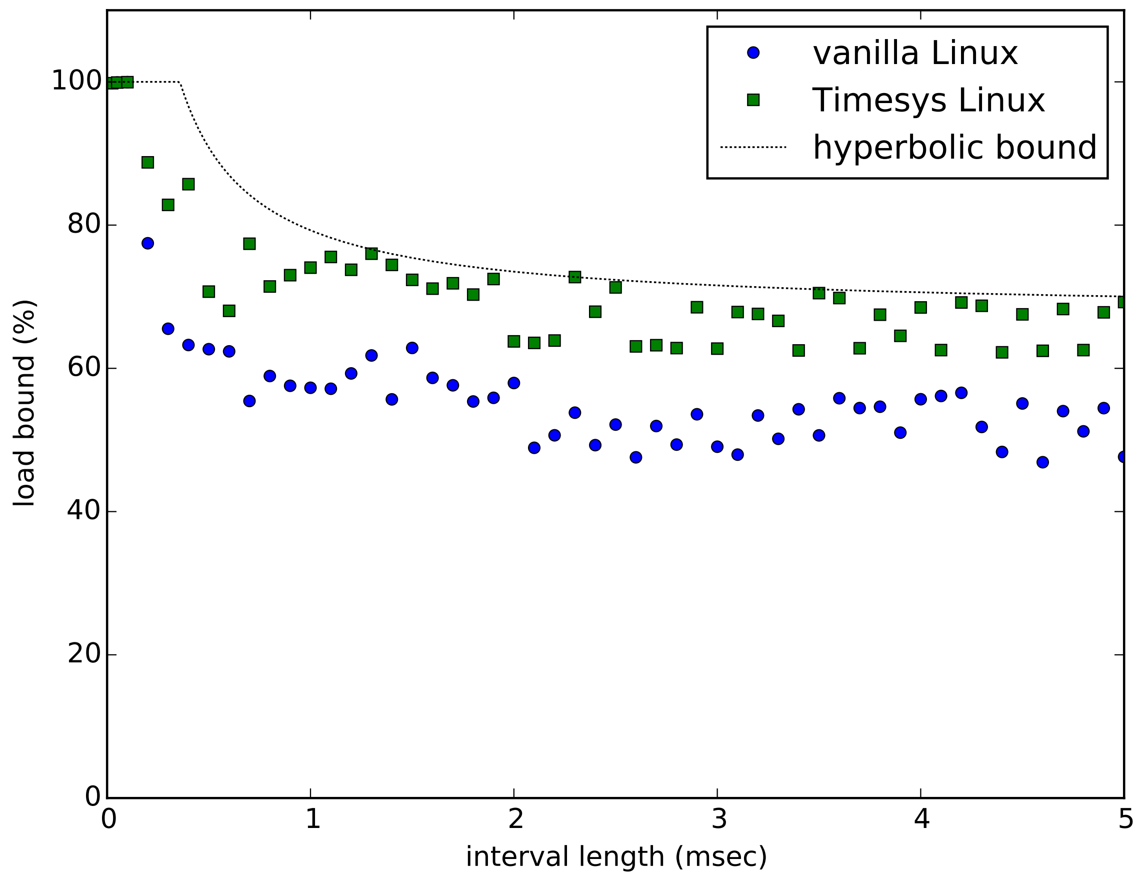


Figure 5.12: Observed interference with ping flooding, including reply.

When enabling NAPI, a reduction in the number of interrupts is expected and as a result a lower average processor interference generated by the network I/O service. The results of enabling and disabling NAPI are shown in Figure 5.13. The benefit of NAPI for vanilla Linux is not immediately obvious from the given data, since there is actually an increase in the processor interference. A possible explanation is that since all softirqs are serviced by the same thread, increasing the priority of the softirq thread allows other activities to run at a higher priority, thereby increasing the measured interference. That is, the measured interference includes activities other than the network processing. Enabling NAPI on the Timesys kernel does appear to result in a slight improvement. This is likely due to the expected reduction in number of interrupts consuming less processor time. Since NAPI has shown to result in better performance and is the default, all experiments, unless explicitly noted otherwise, were run with NAPI enabled.

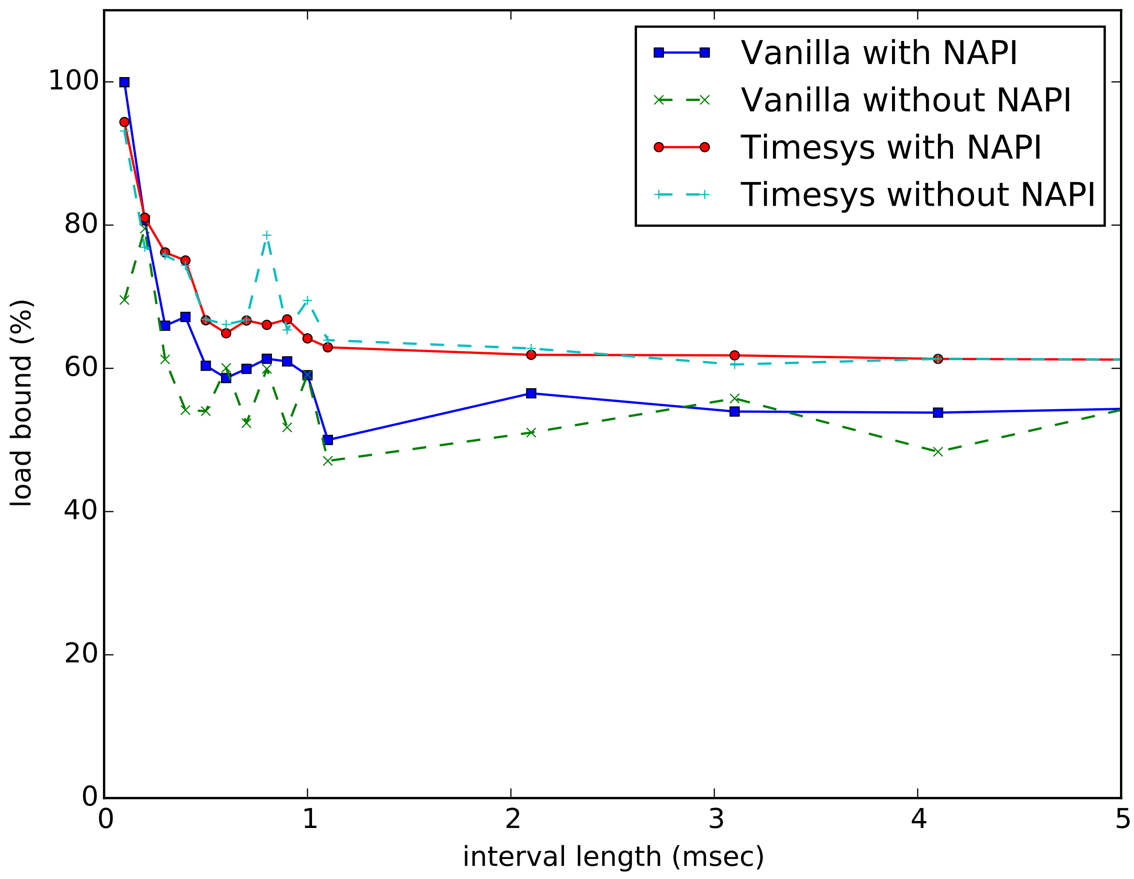


Figure 5.13: Observation of the effects of NAPI on interference.

To carry the analysis further, an experiment was done to separate the load bound for receive processing from the load bound for transmit processing. The normal system action for a ping message is to send a reply message. The work of replying amounts to about half of the work of the network device driver tasks for ping messages. A more precise picture of the interference caused by just the network receiving task can be obtained by configuring the system to not reply to ping requests. The graph in Figure 5.14 juxtaposes the observed interference due to the base operating system; with ping-reply processing, without ping-reply processing, and without any network traffic. The fitted hyperbolic load bounds are also shown for each case. An interesting difference between the data for the "no reply" and the normal ping processing cases is the clear alignment of the "no reply" data into just two distinct hyperbolas, as compared to the more complex pattern for the ping reply case. The more complex pattern of variation in the data for the case with replies may be due to the summing of the interferences of these two threads, whose interference intervals occasionally coincide.

If this is true, it suggests a possible improvement in performance by separating the execution of these two threads.

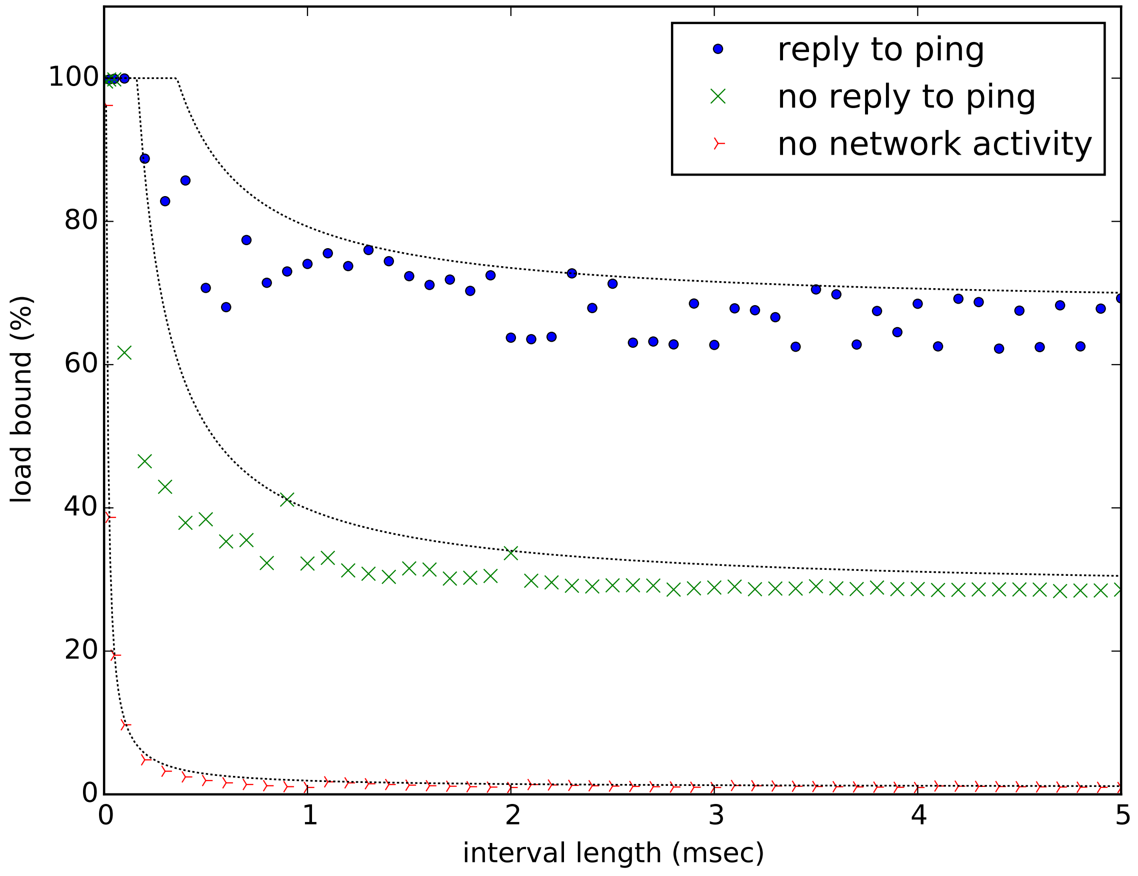


Figure 5.14: Comparing processor interference when ping replies are disabled.

Note that understanding these phenomena (e.g., interaction between send and receive processing) is not necessary to apply the techniques presented in this chapter. In fact, the ability to formulate an abstract workload model *without* precisely understanding the exact mechanisms of the interference is the primary advantage of this technique.

5.5 Relationship between Interference and I/O Service Quality

A system can be configured to force all I/O processing threads to a priority lower than that of all real-time tasks and thereby reduce I/O processor interference nearly to zero. For instance, to verify that the scheduling of the network I/O server threads faithfully honor priorities, an experiment was performed in which the task τ_2 runs at a higher priority than the network I/O server threads. The results are shown in Figure 5.15. It can be seen that the observed interference experienced by τ_2, in this case with a heavy network load, is nearly identical to the observed interference with no network activity. These results provide a demonstration that the network I/O server threads are being scheduled in accordance with the specified priority, which is important since the theoretical schedulability results rely on this assumption.

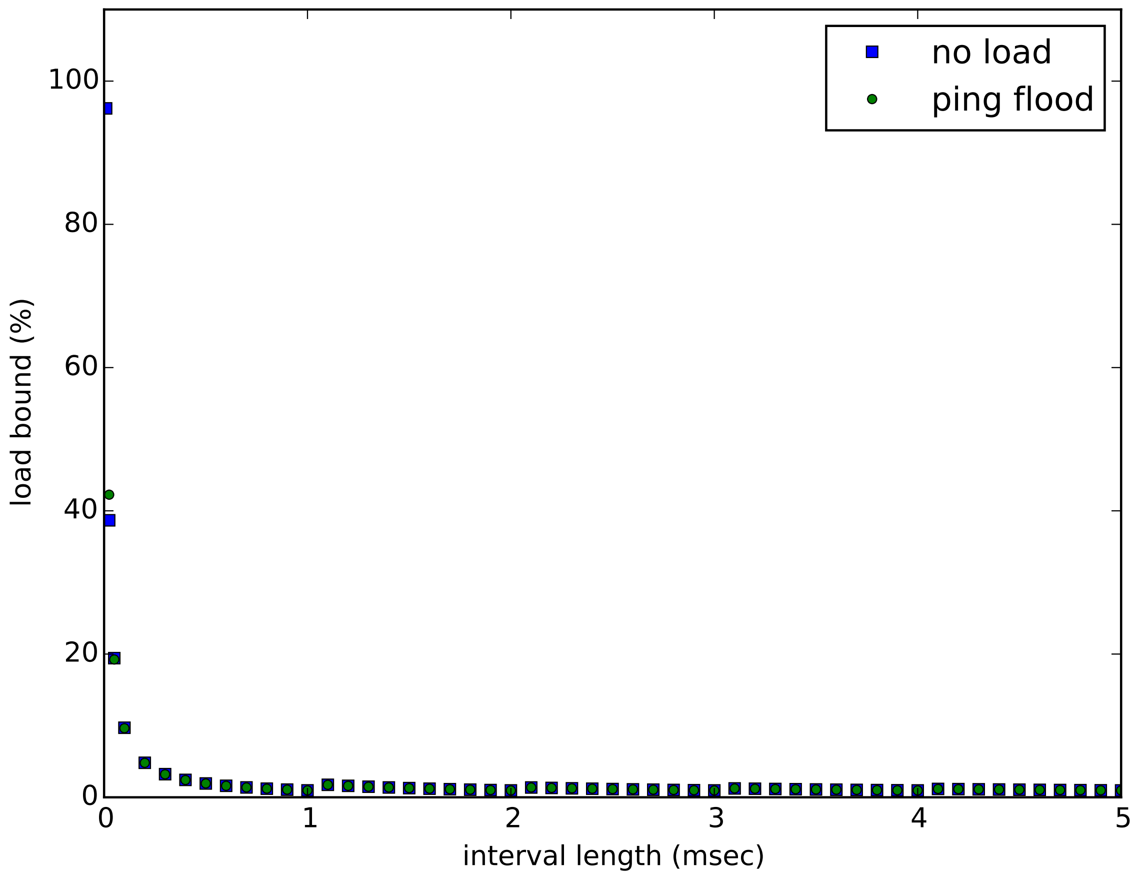


Figure 5.15: Interference measurements with the network I/O thread at a lower priority than the measurement thread.

The problem with configuring the network I/O processing thread with a low priority is that the I/O service quality will likely be affected. For instance, packets considered "high-priority" may incur undesired delays to other higher priority applications. If the network traffic includes data required for a real-time task to operate correctly, then the delayed packets may translate into a failure of such a real-time task.

Table 5.1: Priority assignments of tasks for network I/O quality experiments.

τ_1	τ_2	τ_D	**Linux Version**
high	med	hybrid	vanilla
high	med	low	Timesys
med	low	high	Timesys

The following experiments demonstrate the effectiveness of the derived models from the previous Section 5.4 and also the effect of changing a system's network processing thread priority on network I/O quality. The intuition behind the quality of network I/O service is that the network I/O service quality is proportional to the amount of processor time allocated to its I/O processing thread.

These experiments use three computers and are referred to as Host A, B, and C. Host A sends Host C a "heartbeat" datagram once every 10 msecs. Host B sends ping packets to Host C once every 10μsecs. Host C also executes the following SCHED_FIFO scheduled threads:

- τ_D signifies the network I/O service threads that execute on the system processor to send and receive network packets. The packets are transferred to/from the network interface card and also to/from applications through the processing performed with these threads. On Timesys Linux, there are two kernel threads *softirq-net-rx* and *softirq-net-tx*. On vanilla Linux, the packet processing takes place in softirq threads, which may execute in interrupt context or a schedulable thread.

- τ_1 is a periodic task with a period and relative deadline of 10 msecs and a worst-case execution time of 2 msecs. The final operation of each job in τ_1 is a non-blocking network receive operation for a periodic "heartbeat" packet (UDP datagram). The heartbeat packet loss rate is used to measure the quality of network I/O service.

- τ_2 is also a periodic task with a period and relative deadline of 10 msecs. The execution time and priority relative to τ_D varies across the following experiments. The number of missed deadlines experienced by τ_D is used to demonstrate the interference effects from τ_D.

τ_1 and τ_2 were implemented with a modified Hourglass benchmarking program [93] to provide τ_1's non-blocking network receive operations. Data for comparing the quality of I/O service was gathered using various priority configurations detailed in Table 5.1. Figures 5.16 and 5.17 show the differences between the configurations in terms of missed deadlines and heartbeat packets respectively.

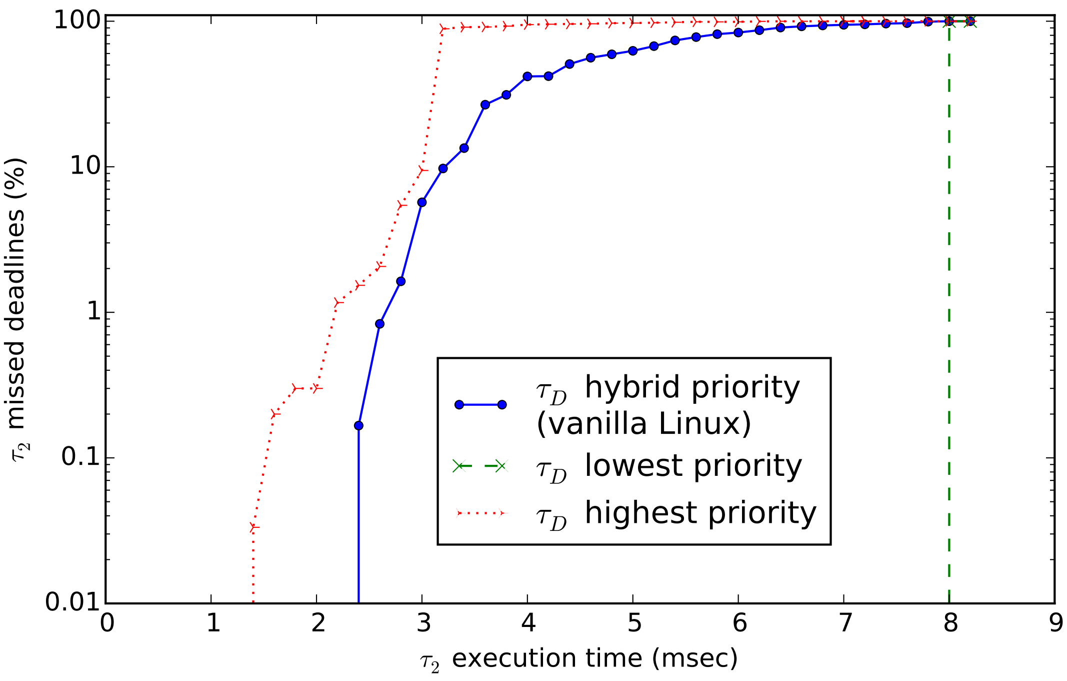


Figure 5.16: Missed deadline percentage of τ_2 ($e_2 = various$, $p_2 = 10$) with interference from τ_1 ($e_1 = 2$, $p_1 = 10$) and τ_D subject to one PING message every 10 μsec.

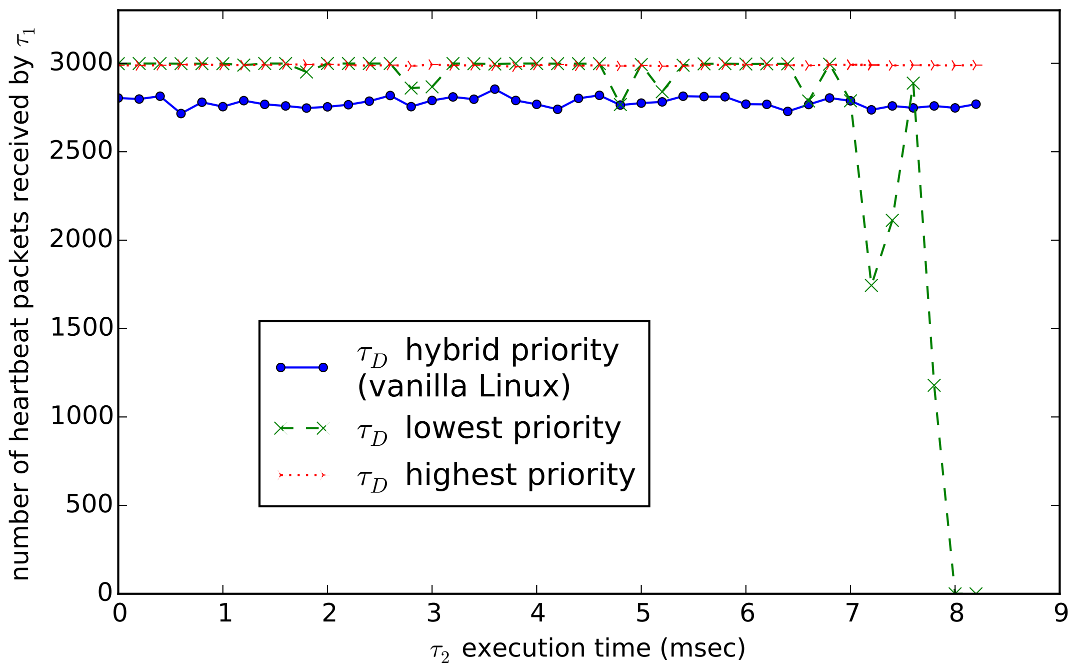


Figure 5.17: Number of heartbeat packets received by τ_1 ($e_1 = 2$, $p_1 = 10$) with interference from τ_2 ($e_2 = various$, $p_2 = 10$) and τ_D subject to one PING message every 10 μsec.

The Traditional (vanilla) Linux two-level (hybrid) priority configuration results in τ_2 missing deadlines at lower utilizations than the other priority configurations. Also, a higher relative heartbeat packet loss rate is also experienced.

The Background Server experiments confirm that assigning the network packet processing to the lowest relative priority maximizes the number of met deadlines by τ_2. However, this configuration has the adverse effect of increasing the number of lost packets. Figure 5.18 shows the combined load of τ_1 and τ_2. The values near the deadline (10 msecs) suggest that if there is no interference from τ_D or other system activity, τ_2 is expected to complete within its deadline until e_2 exceeds 7 msec. This is consistent with the data in Figure 5.16. The heartbeat packet receipt rate of τ_1 starts out better than vanilla Linux but degenerates as the execution time of τ_2 (e_2) increases due to insufficient CPU time.

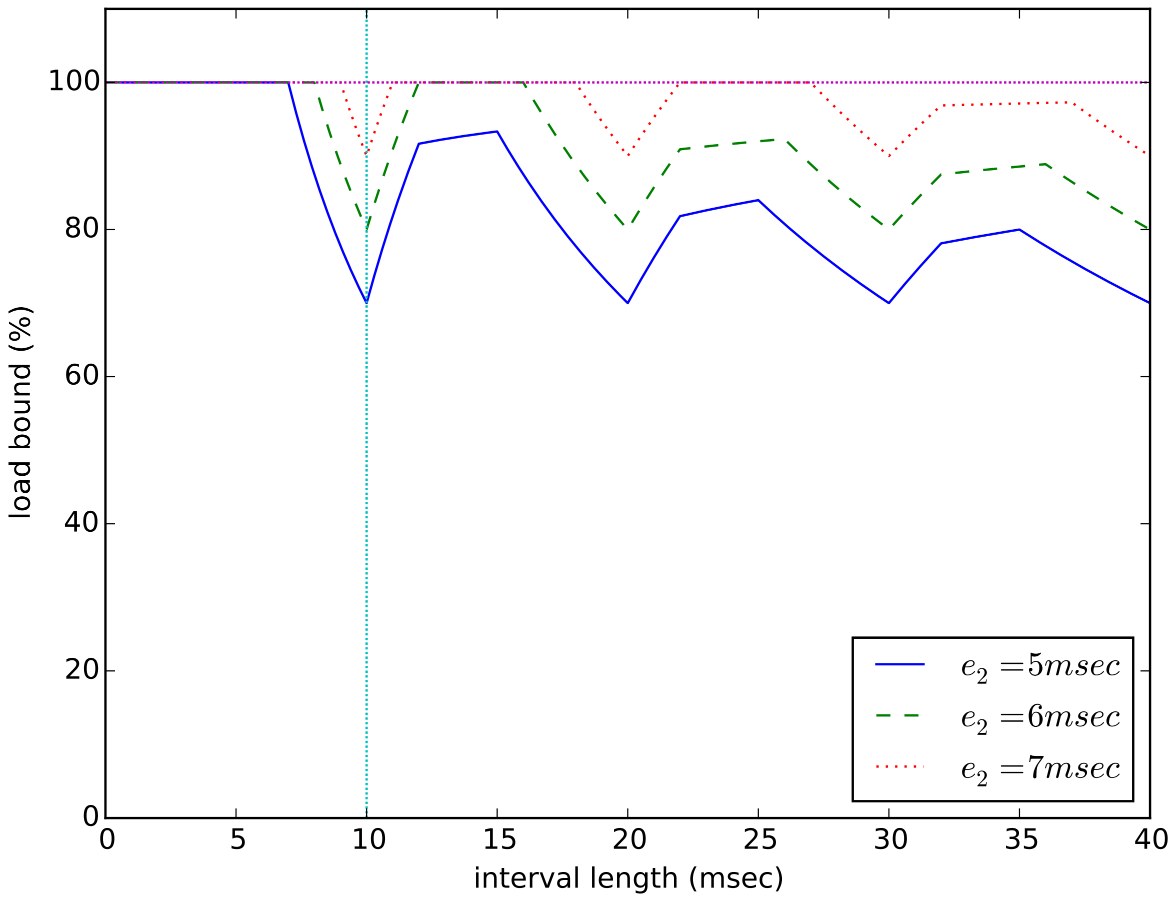


Figure 5.18: Sum of load-bound functions for τ_1 ($e_1 = 2$, $p_1 = 10$) and τ_2 ($e_2 = various$, $p_2 = 10$), for three different execution time values of τ_2.

The Foreground Server experiments confirms our expectation that assigning the highest priority to τ_D results in the best heart-beat packet reception performance, but results in τ_2 missing the largest number of deadlines. The line labeled $\tau_1 + \tau_D$ in Figure 5.19 shows the sum of the theoretical load bound of τ_1 and the empirically derived hyperbolic load bound of τ_D (Section 5.4). By examining the graph at the deadline (10 msec), and allowing some margin error (e.g., release-time jitter, overhead and measurement errors), a logical prediction is that τ_2 should not miss any deadlines until its execution time exceeds 1.2 msec. This appears to be consistent with the actual performance in Figure 5.16.

The results for the priority configurations described above and detailed in Table 5.1 show that meeting all deadlines and receiving all packets is not simultaneously achievable for many system states. A potential solution is to limit the CPU usage of τ_D such that it can only receive CPU time up to the amount that would not cause missed deadlines. As such, a larger set of task configurations (e.g., execution time of τ_2) could be supported. However, neither the vanilla nor Timesys kernel have a scheduling policy to provide such a limit. The following Chapter 6 discusses the use of several aperiodic server scheduling algorithms to limit the CPU usage.

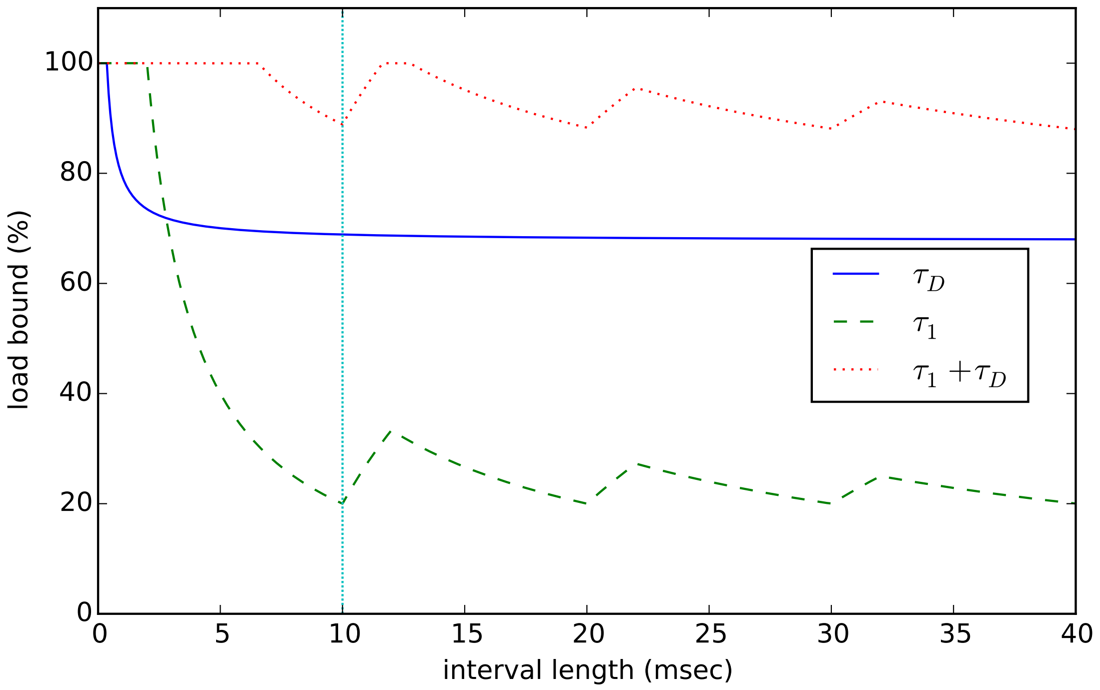


Figure 5.19: Individual load-bound functions for τ_1 ($e_1 = 2$, $p_1 = 10$) and τ_D, along with their summed load bounds.

5.6 Conclusion

This chapter presents a novel technique to derive a theoretical workload model for I/O-service processor demand using empirical measurements. Real-time scheduling theory relies on well-defined abstract workload models, however, I/O-service processor usage on GPOSs is generally not implemented to conform to such workload models. Therefore, without abstract workload models a large body of schedulability analyses cannot be effectively used. The solution presented in this chapter is to perform measurements of specific time intervals under heavy load conditions in order to derive load bound abstract workload model for I/O-service that can be used with theoretical schedulability analyses.

The fixed-task priority that the I/O-service processor thread is scheduled impacts the quality of the I/O-service provided to applications. Intuitively, the lower priority of the I/O thread in relationship to other tasks results in less timely processor allocation to service I/O. The effect on quality is illustrated in this chapter through network I/O experiments, where the lower priority network processor thread translates into larger response times and dropped packets due to insufficient buffers to store unprocessed packets.

Setting the I/O processor thread to a specific priority tends be insufficient many desired system designs. For instance, any threads set at a lower priority in relation to the I/O thread may starved, depending on the I/O load. Such lower priority threads may have timing constraints and starvation may result in missed deadlines. However, setting the I/O thread to a low priority may starve the I/O resulting in dropped packets and missed deadlines for users of the I/O service.

A potential solution is to schedule the I/O thread using an aperiodic server scheduling algorithm, which would allow a system designer to bound the amount of processor time the I/O service can consume at a given priority. Such bounding of processor time is desired for configuring the I/O service appropriately with respect to other application deadlines. Further, given that the abstract model for I/O processor usage is derived from empirical measurements, using an aperiodic server scheduling algorithm provides a extra level of assurance that other tasks on the system will not be affected if the derived bound is not valid (e.g., the actual worst-case were not encountered and subsequently measured). Therefore, the following Chapter 6 uses the derived model presented in this chapter and improves the flexibility of incorporating I/O service with an aperiodic server scheduling algorithm to (1) allow one to force the derived model and (2) fine-time the I/O processor time.

CHAPTER 6

BALANCING PROCESSOR LOAD AND I/O SERVICE QUALITY

This chapter discusses the application of the *sporadic server* scheduling algorithm [106] to allocate processor time for I/O service. The motivation behind using sporadic server is to provide the ability to actively control the amount of processor time allocated to I/O services. Rather than being limited to a single load-bound function as derived in the previous Chapter 5, a sporadic server can be used to vary (e.g., reduce processor interference) the processor usage of I/O services in order support more diverse sets of real-time tasks.

My research contributions, discussed in this chapter, include modifications to the sporadic server scheduling algorithm to correctly and efficiently account for practical factors of an implementation on real-world hardware. In particular, my contributions are:

- approach for handling overruns of allocated processor time
- corrections to the POSIX version of sporadic server to allow its processor allocation to be modeled as a periodic or sporadic task
- design allowing one to limit the amount of interference a sporadic server can cause on non-real-time tasks
- technique to balance low latency and high throughput for I/O service by adjusting the amount of incurred processor overhead

A large portion of this chapter's results were the result of a collaborative effort among Theodore Baker, Andy Wang, Michael González Harbour, and me. The results were published in [108, 109].

6.1 Introduction

Requests for I/O processor time are generally much different than periodic tasks. The requests for I/O processor service time is often not very well-defined, especially over short time intervals. For instance, suppose packets are sent periodically from a given machine A to another machine B.

The reception of packets by machine B (received by the network interface card) from A will likely not be periodic. Delays due collisions, buffering, etc. can result in bursts of network activity on Host B. These bursts can result in large intervals of continuous processor usage potentially starving other real-time activities for extended time intervals, resulting in missed deadlines.

Aperiodic server scheduling algorithms are one method to control the allocation of processor time to non-periodic activities. Rather than assigning a fixed priority to a given system activity, the processor allocation is constrained by the aperiodic server to actively enforce processor usage at a given priority to mimic that of a periodic task.

Sporadic server is an aperiodic server scheduling algorithm that was chosen for this dissertation research as a good candidate to schedule I/O service processor threads. The theoretical model of sporadic server results no greater interference than a periodic task with equivalent parameters and generally provides better average-case response time than other fixed-task priority aperiodic servers. Further, sporadic server has been standardized by the POSIX specification.

The following discusses the practical implications for an implemented sporadic server. First, creating a implementation that correctly maintains the property that the interference is no worse than an equivalent periodic task is not straightforward. In fact, the POSIX specification is shown to be incorrect in this regard. This dissertation provides fixes for these errors in the specification, proposes an improved design to reduce the impact on non-real-time applications, and finally improves the efficiency for using sporadic server to schedule network I/O service processor threads.

6.2 From Theory to Implementation

During the late 1980's and early 1990's, a major initiative was undertaken to disseminate then-recent technological developments in real-time systems through programming language and operating system standards. One success of this effort was the inclusion of support for preemptive fixed-task-priority scheduling policies in the IEEE POSIX standard application program interface (API) for operating system services. That standard has since been rolled into the Single UNIX Specification of The Open Group [43] and is implemented by Linux and many other operating systems. However, as advances have continued to be made in the understanding of real-time scheduling, very little has been done to update the POSIX standard.

In this chapter, corrections to the existing standards are given and the case is made for the need to correct the SCHED_SPORADIC scheduling policy specification in the existing POSIX real-time scheduling standard. The following dissertation research shows that the current specification has several critical technical flaws, argues for the importance of correcting these flaws, and provides specific suggestions for how they may be corrected.

The SCHED_SPORADIC policy is important because it is the only scheduling policy supported by the POSIX standard that enforces an upper bound on the amount of high-priority execution time that a thread can consume within a given time interval. As such, it is the only standard scheduling policy that is potentially suitable for compositional schedulability analysis of an "open" real-time system in the sense of [25], and the only one that is suitable as the basis for a virtual computing resource abstraction for compositional analysis of a hierarchical scheduling scheme such as those studied in [16, 23, 66, 98, 105, 117].

The SCHED_SPORADIC policy is a variation on the *sporadic server* scheduling concept, originally introduced by Sprunt, Sha, and Lehoczky [106]. Conceptually, a sporadic server has execution time *budget*, which it consumes while it executes at a given *server priority*, and which is *replenished* according to a rule that approximates the processor usage of a conceptual set of sporadic tasks with a given *period*. The intent is that the worst-case behaviors of the server – both the minimum level of service it provides and the maximum amount of processor time it consumes – can be modeled by an equivalent periodic task, whose worst-case execution time is equal to the server budget and whose period is equal to the server period. For this dissertation, the property is referred to as the *periodic task analogy*.

This alleged equivalence of a sporadic server to a periodic task is often cited in the literature. For example, [23] says that "Sprunt proved that in the worst-case the interference due to a Sporadic Server is equivalent to that of a simple Periodic Server", and [98] says "in the worst case, a child reserve [implemented as a sporadic server] behaves like a classical Liu and Layland periodic task".

Unfortunately, the original formulation of the sporadic server scheduling algorithm published in [106] – commonly called the *SpSL* sporadic server – violates the above assertions. A defect in the replenishment rules allows a thread to consume more processor time than the allegedly-equivalent periodic task. It is not known for certain who first discovered this defect. Michael González

Harbour, cited as a source in [13], stated that he first learned of it from Raj Rajkumar in personal communication. The defect is also described in [68].

Several proposals for correcting this defect have been published, including one in [13], several variations in [68], and an adaptation for deadline scheduling in [33]. In particular, the formulation of the sporadic server scheduling policy in the POSIX standard was widely believed to have corrected this defect. For example, [13] says: "The POSIX sporadic server algorithm (PSS) provides an effective and safe solution that does not allow any budget overruns".

Believing the POSIX sporadic server to be correct, improvements to the previous chapter's research through actively scheduling the I/O service processor time was explored. Earlier work proposed that the network I/O processing of incoming and outgoing network traffic be executed by a thread that is scheduled using the SCHED_SPORADIC policy. A sporadic server (SS) implementation was created by Mark Lewandowski for our publication [61], which used the kernel's periodic tick to schedule processor time. The preliminary results using his implementation suggested that using finer-grained processor time management would improve system performance. However, rather than modifying Lewandowski's SS implementation, initial experiments were conducted with a newly published SS implementation, using the Linux kernel's high resolution timers, written by Dario Faggioli, and posted to the Linux kernel mailing list [29]. After making refinements to Faggioli's SS implementation in order to adhere to the POSIX SS specification, I was able to repeat Lewandowski's prior experiments. In the results from these follow-up experiments, it was surprising to see that the server's actual processor utilization was significantly higher than that of a periodic task with the same budget and period. Looking for the cause of this anomalous behavior, led to the discovery of two flaws in the POSIX specification.

To that end, the first part of this chapter demonstrates the following facts:

1. The POSIX sporadic server algorithm's replenishment rules suffer from an effect that this dissertation calls "premature replenishment". Through experimental results, it is shown that this defect allows a server to use an average of 38 percent more execution time than an analogous periodic task.

2. The POSIX sporadic server algorithm also suffers from an unreported defect, which this dissertation calls "budget amplification". This defect allows a server to consume an amount of processor time arbitrarily close to 100 percent, regardless of the size of the server's budget.

3. These defects can be corrected by modifications to the POSIX sporadic server specification, which are described in this chapter.

In support of the above, empirical data is provided with an implementation of the POSIX sporadic server in the Linux kernel, which clearly demonstrate the interference effect of budget amplification on a periodic task. Further, simulation results using pseudo-random job arrivals are provided to give some insight into the likelihood of encountering the effects of the above two defects in practice.

Additionally, this dissertation research proposes an additional modification to the POSIX sporadic server specification to address a practical deficiency relating to the inability to sufficiently lower the priority of a sporadic server when it is out of budget. In particular, this proposal addresses the inability to configure the amount of interference sporadic server can cause for any application (including non-real-time) running on the processor, not just those applications set with real-time priorities.

6.3 An Ideal Sporadic Server Model

As discussed in Chapter 3, the preemptive scheduling and of periodic task systems is well understood and has been studied extensively, starting with the pioneering work of [67] and the recursive response-time analysis technique of [59].

A *periodic server* is a mechanism for scheduling an aperiodic workload in a way that is compatible with schedulability analysis techniques originally developed for periodic task systems. Aperiodic requests (jobs) are placed in a queue upon arrival. The server *activates* at times $t_1, t_2, \ldots$ such that $t_{i+1} - t_i = T_s$, where T_s is the nominal server *period*, and executes at each activation for up to C_s, where C_s is the server *budget*. If the server uses up its budget it is preempted and its execution is suspended until the next period. If the server is scheduled to activate at time t and finds no queued work, it is deactivated until $t + T_s$. In this way the aperiodic workload is executed in periodic bursts of activity; i.e., its execution is indistinguishable from a periodic task.

A *primitive sporadic server* is obtained from a periodic server by replacing the periodic constraint $t_{i+1} - t_i = T_s$ by the *sporadic constraint* $t_{i+1} - t_i \geq T_s$. That is, the period is interpreted as just a lower bound on the separation between activations. The sporadic constraint guarantees that the worst-case processor interference caused by a sporadic task for other tasks is not greater than that caused by a periodic task with the same worst-case execution time and period. In other words, the processor demand function (and therefore a worst-case residual supply function for other tasks) of the server will be no worse than a periodic task with period T_s and worst-case execution time C_s. In this way, the periodic task analogy holds.

A primitive sporadic server has an advantage over a periodic server due to *bandwidth preservation*; that is, it is able to preserve its execution time budget under some conditions where a periodic server would not. If there are no jobs queued for a sporadic server at the time a periodic server would be activated, the sporadic server can defer activation until a job arrives, enabling the job to be served earlier than if its service were forced to wait until the next period (i.e., activation time instant) of the periodic server.

An *ideal sporadic server* is a generalization based on a conceptual swarm of unit-capacity sporadic tasks, called "*unit servers*" or just "*units*", for short. The basis for this generalization is the observation that the worst-case analysis techniques of [67] and [59] allow a set of periodic or sporadic tasks with the identical periods to be treated as if they were a single task, whose execution time is the sum of the individual task execution times. That is, the worst-case interference such a swarm of identical sporadic tasks can cause for other tasks occurs when all the tasks are released together, as if they were one task. Although the worst-case interference for lower-priority tasks caused by such a swarm of sporadic servers remains the same as for a single periodic server task, the average response time under light workloads can be much better. Indeed, studies have shown that sporadic servers are able to achieve response times close to those of a dedicated processor under light workloads, and response times similar to those of a processor of speed $u_s = C_s/T_s$ under heavy loads.

Since the overhead (e.g., time switch between tasks) of implementing a server as a swarm of literal unit-capacity sporadic servers would be very high, published formulations of sporadic server scheduling algorithms attempt to account for processor capacity in larger chunks of time, called *replenishments*. Each replenishment R may be viewed as representing a cohort of $R.amt$ unit servers that are eligible to be activated at the same replenishment time, $R.time$. For such a sporadic server formulation to satisfy the periodic task analogy, the rules for combining unit servers into replenishments must respect the sporadic constraint.

Observation 1

If R represents a cohort of unit servers that were activated together at some time t and executed *during a* busy interval *containing t,*[1] *the sporadic constraint will be satisfied so long as $R.time \geq t + T_s$.*

Observation 2

The sporadic constraint is preserved if $R.time$ is advanced to any later time.

Observation 3

The sporadic task constraint is preserved if a replenishment R_1 is merged with a replenishment of R_2 to create a replenishment R_3 with $R_3.amt = R_1.amt + R_2.amt$ and $R_3.time = R_1.time$, provided that $R_1.time + R_1.amt \geq R_2.time$.

Proof

Suppose cohorts corresponding to R_1 and R_2 are activated at $R_1.time$. Since unit servers are indistinguishable within a cohort, we can assume that those of R_1 execute first and so cannot complete sooner than $R_1.time + R_1.amt$. Since $R_1.time + R_1.amt \geq R_2.time$, by the time R_1 completes the replenishment time t_2 will have been reached. So, none of the unit servers in the combined R_3 can activate earlier than if R_1 and R_2 are kept separate. $\square$

6.4 POSIX Sporadic Server Deficiencies

The POSIX sporadic server policy specified in [43] superficially resembles the ideal model described in Section 6.3. A thread subject to this policy has a native priority, specified by the parameter *sched_priority*, a budget C_s specified by the parameter *sched_ss_init_budget*, and a period T_s specified by the parameter *sched_ss_repl_period*. The thread has a numerical attribute, called the *currently available execution capacity*, which abstracts a set of unit servers that are eligible for activation (because their most recent activations are all at least T_s in the past), and a set of pending replenishments, which abstracts sets of unit servers that are not yet eligible for activation (because the last activation is less than T_s). If the POSIX specification were in agreement with the ideal model, each replenishment R would correspond to a cohort of units that executed within a busy interval of the server and *R.time* would be earliest time consistent with Observation 1. However, the POSIX rules for handling replenishments fail to enforce the sporadic constraint at the unit server level, and so break the periodic task analogy.

[1]A *busy interval* is a time interval during which the processor is continuously executing the server and tasks that the server cannot preempt. The processor is not idle during any time interval in a busy interval.

In this section we compare the POSIX sporadic server policy [43] and its terminology to the ideal model described previously. These comparisons will be used to explain how the resulting defects occur.

6.4.1 Budget Amplification

POSIX differs from the ideal model by limiting a server's execution "to at most its available execution capacity, *plus the resolution of the execution time clock used for this scheduling policy*". Such allowance for inexact execution budget enforcement is essential in a practical implementation. Typically enforcement of budget can vary from zero to the maximum of the timer latency and the longest non-preemptible section of the system calls that a server may perform. POSIX errs in stating that when "the running thread ... reaches the limit imposed on its execution time ... the execution time consumed is subtracted from the available execution capacity (*which becomes zero*)." The specified one-tick enforcement delay stated above allows the server budget to become negative by one tick, and in reality, larger overruns are likely. POSIX handles these overruns elsewhere by stating that "when the running thread with assigned priority equal to *sched_priority* becomes a preempted thread ... and the execution time consumed is subtracted from the available execution capacity ... If the available execution capacity would become negative by this operation ... *it shall be set to zero*". POSIX attempts to compensate for the downstream consequences of forgiving such overruns by specifying that if as a result of a replenishment "the execution capacity would become larger than *sched_ss_initial_budget*, it shall be rounded down to a value equal to *sched_ss_initial_budget*." However, this is an oversimplification, which cannot be translated into the ideal model.

This oversimplification of the ideal model leads to the defect referred to in this dissertation as *budget amplification*. That is, the size of a replenishment can grow as it is consumed and rescheduled over time.

When an overrun occurs, the POSIX specification states that the available execution capacity should be set to zero and that a replenishment should be scheduled for the amount of the time used since the *activation_time*. At this point, the sporadic server has used more units than it had available and schedules it as a future replenishment. This scheduling decision would result in more interference than the length of time for 1 tick as long as the sporadic server were charged for this amount of time. However, setting the execution capacity to zero means that the overrun amount is never charged, thereby increasing the total capacity the server can demand within its period.

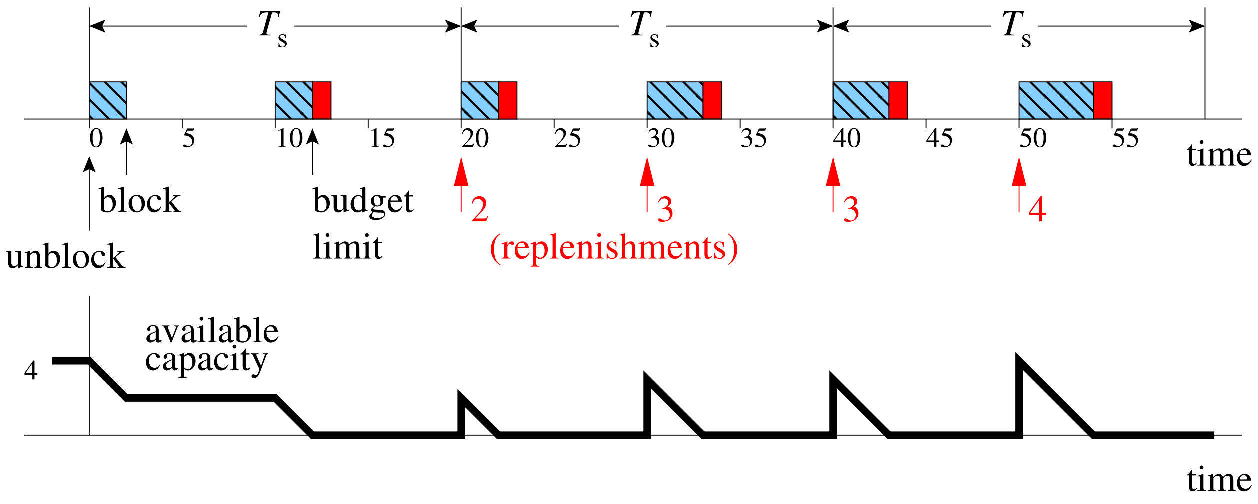


Figure 6.1: Budget amplification anomaly.

While POSIX attempts to diminish the effect of overruns by rounding the currently available execution capacity down to the initial budget, this effort is not sufficient. Consider the example illustrated in Figure 6.1. Here the resolution for the execution time clock is 1 time unit. At time 0, the server executes for two time units and schedules a replenishment of two time units at time 20. At time 10, the server again begins execution, but at time 12 it has not completed executing and therefore is scheduled to stop running at its high priority. The server is able to execute an additional time unit before actually being stopped, as permitted in the POSIX specification. At time 13, a replenishment is scheduled at time 30 for the amount of capacity consumed, which in this case is 3, and the available execution capacity is set to zero. Now, the sum of pending replenishments is greater than the initial budget of 4, but within the leeway provided by the specification. This sequence of receiving one time unit of additional execution capacity repeats with the intervals of server execution beginning at 20, 30, 40, and 50. By the time the replenishment for the execution interval beginning at 30 is scheduled, the sum of pending replenishments is 2 time units greater than the initial budget. If this scenario continues each overrun will contribute to an increase of the total execution time available to the sporadic server. As long as each replenishment is below the maximum budget, this amplification may continue. In this case, each replenishment can grow to at most 5 time units (4 due to the initial budget limit and 1 for the permitted clock resolution).

With this defect, by breaking the budget into small enough fragments a server can achieve an execution capacity arbitrarily close to 100%.

6.4.2 Premature Replenishments

POSIX specifies that "a replenishment operation consists of adding the corresponding *replenish_ amount* to the available execution capacity at the scheduled time". This has the effect of maintaining a single activation time for all currently available units. This is inconsistent with the ideal model, because it fails to preserve the minimum replenishment time (earliest next activation time) of a replenishment (cohort of server units) if the server is in a busy period when a replenishment arrives. A consequence is that a replenishment can arrive earlier than its required minimum offset from the previous arrival, resulting in what we refer to as a premature replenishment.

Table 6.1: Periodic task set for premature replenishment example.

Task	$\mathbf{C_i}$	$\mathbf{T_i}$	$\mathbf{D_i}$
τ_1	10	200	20
τ_2	20	50	50
τ_3	49	200	100

The following example illustrates the premature replenishment defect. Consider a scenario with three independent periodic tasks, given a deadline-monotonic priority ordering and parameters (worst-case execution time, period, relative deadline) shown in Table 6.1. Response time analysis [59] obtains a worst-case response time for task τ_3 of 99 time units:

$$R_3 = \left\lceil \frac{99}{200} \right\rceil C_1 + \left\lceil \frac{99}{50} \right\rceil C_2 + C_3 = 10 + 2 \cdot 20 + 49 = 99$$

Suppose task τ_2 is a sporadic server serving aperiodic events. The sporadic server is given an execution capacity $C_2 = 20$, and a replenishment period $T_2 = 50$. Under the ideal sporadic server model the worst-case response time of τ_3 would be 99. However, in the execution sequence shown in Figure 6.2, the response time of τ_3 is 117 (and therefore its deadline is missed). In this sequence, aperiodic events arrive at times $\{0, 40, 90\}$, with respective execution-time demands of $\{18, 20, 20\}$. Task τ_3 is activated at $t = 0$, while task τ_1 is activated at $t = 41$. We can see that when the second aperiodic event arrives at $t = 40$, the execution capacity of the sporadic server is above zero (its value is 2), so the activation time is recorded as 40, and the aperiodic event starts to be processed. At time 41, τ_1 preempts the execution of the sporadic server. When the replenishment of the first chunk of execution time occurs at $t = 50$, 18 is added to the available execution capacity (1 unit at that time), and the activation time remains unchanged (because the server is still active). This violates the ideal model, by effectively merging a cohort of 18 units not permitted to activate until time 50 with a cohort of two units that activated at time 40. When the aperiodic event is fully processed, a replenishment of 20 time units is scheduled prematurely to happen at $t = 90$, based on the activation time at time 40. This allows the service of three long aperiodic events to preempt task τ_3, instead of the two that would happen in the ideal model.

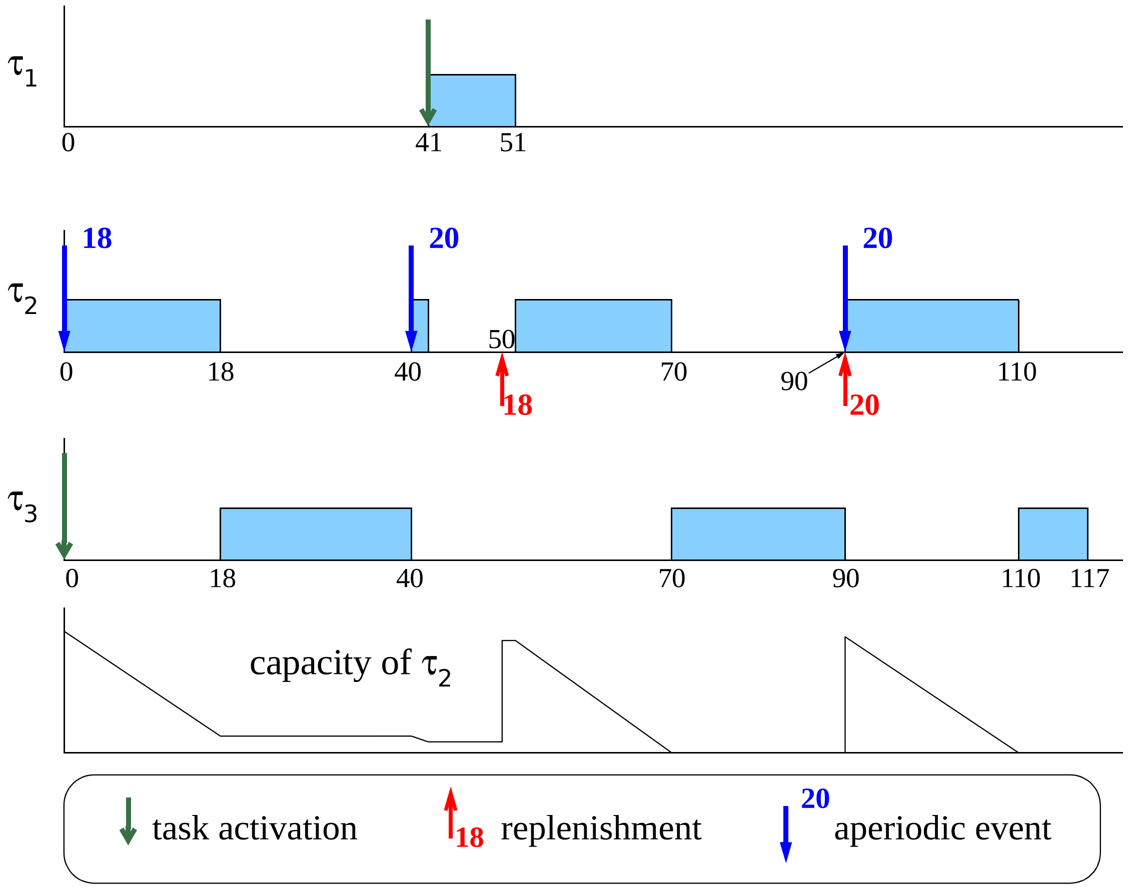


Figure 6.2: Execution sequence showing a replenishment that occurs prematurely.

6.4.3 Unreliable Temporal Isolation

In many real-time systems there is a need to provide *temporal isolation* between tasks or between sub-systems. That is, when one composes subsystems one wants a guarantee that if a task in one subsystem fails to complete within its allotted time budget it cannot cause a task in another subsystem to miss a deadline. As mentioned in the Introduction, sporadic server scheduling has been proposed in a number of papers on compositional and hierarchical scheduling for “open” real-time systems, as a means of achieving temporal isolation.

The theoretical formulation of the Sprunt sporadic server [106] provides temporal isolation by requiring that when a task runs out of budget, it will not execute until its budget is replenished. The POSIX formulation differs in allowing a sporadic server to continue execution after it has exhausted its budget, albeit at a lower (*background*) priority, specified by the parameter *sched_ss_low_priority*. The apparent intent behind this feature is to allow a server to make use of otherwise-idle time. This feature is compatible with the ideal model so long as *sched_ss_low_priority* is below the priority of every critical task. However, POSIX also specifies that each scheduling policy has a range of valid priorities, which is implementation defined. Further, the statement that the SCHED_SPORADIC policy “is identical to the SCHED_FIFO policy with some additional conditions” has been interpreted by some to mean that the range of priorities for these two policies should be the same. For example, in Linux the priorities for SCHED_FIFO, SCHED_SPORADIC, and SCHED_RR are identical, while priorities for SCHED_OTHER are strictly lower. This means that a thread under any real-time policy can lock out all SCHED_OTHER threads, breaking temporal isolation.

The problem of applications scheduled using real-time priorities has been recognized and an attempt to alleviate this problem has been implemented in Linux through *real-time throttling* [21, 123], at the expense of breaking POSIX compliance. Real-time throttling ensures that in a specified time period, the non-real-time threads receive a minimum amount of processor time. Once the budget for all real-time threads is consumed, the processor is taken away from the real-time threads to provide processor time to the threads scheduled with SCHED_OTHER (non-real-time) priorities. This mechanism prevents real-time threads from locking up the system, but it is rather coarse. There is only one budget and period, defined system wide. The default budget is 950 msec of real-time execution time per 1 second period. This means that any real-time thread can experience a (rather large and possibly fatal) preemption of 5 msec.

6.5 Corrected Sporadic Server Algorithm

In this section a corrected version of the POSIX sporadic server is presented, followed by explanations as to how this new version corrects the defects mentioned previously.

Each sporadic server instance S has a replenishment queue $S.Q$, which may contain a maximum of $S.max\_repl$ replenishments. Each replenishment R has time $R.time$ and an amount $R.amt$. The queue $S.Q$ is ordered by replenishment time, earliest first. The sum of the replenishment amounts is equal to the server's initial budget

$$S.budget = \sum_{R \in S.Q} R.amt$$

A server is in foreground mode, competing for processor time at $S.foreground\_priority$ or in background mode, competing at $S.background\_priority$. Whenever S is in foreground mode, its execution time is accumulated in the variable $S.usage$. The currently available execution capacity of the server is computed as

$$S.capacity = \begin{cases} 0 \text{ if } S.Q.head.time > Now \\ S.Q.head.amt - S.usage \text{ otherwise} \end{cases}$$

S is in foreground mode whenever $S.capacity > 0$, and should call Budget_Check as soon as possible after the system detects that $S.capacity \leq 0$ (the server may change to background mode). To detect this condition promptly, event $S.exhaustion$ is queued to occur at time $Now + S.capacity$ whenever S becomes a running task at its foreground priority. The system responds to event $S.exhaustion$ by updating $S.usage$ with the amount of execution time used at the foreground priority since the last update and then executing Budget_Check (Figure 6.3).

The system also calls Budget_Check when S is executing in foreground mode and becomes blocked or is preempted, after cancellation of event $Q.exhaustion$. If S blocks while in foreground mode, after Budget_Check the system executes procedure Split_Check (Figure 6.4).

If S goes into background mode while it is not blocked (in Budget_Check) or if it becomes unblocked while in background mode (in Split_Check) event $S.replenishment$ is queued to occur at time $S.Q.head.time$. The system responds to event $S.replenishment$ by setting $S.priority$ to $S.foreground\_priority$, which may result in S being chosen to execute next.

If S becomes unblocked the system executes procedure Unblock_Check, shown in Figure 6.5.

6.5.1 Correcting for Budget Amplification

In [33] a solution for blocking effects caused by a deadline sporadic server is provided, where overruns will be charged against future replenishments. This mechanism is adapted to allow our modified sporadic server to handle overruns properly and inhibit the budget amplification effect.

```
Budget_Check

 1  if $S.Capacity \leq 0$ then
 2     while $S.Q.head.amt \leq S.usage$ do
          ▷ Exhaust and reschedule the replenishment
 3        $S.usage \leftarrow S.usage - S.Q.head.amt$
 4        $R \leftarrow S.Q.head$
 5        $S.Q.pop$
 6        $R.time \leftarrow R.time + S.Period$
 7        $S.Q.add(R)$
 8     if $S.usage > 0$ then  ▷ $S.usage$ is the overrun amt.
          ▷ Budget reduced when calculating $S.capacity$
          ▷ Due to overrun delay next replenishment
          ▷ Delay cannot be greater than $S.Q.head.amt$ (while loop condition)
 9        $S.Q.head.time \leftarrow S.Q.head.time + S.Usage$
          ▷ Merge front two replenishments if times overlap
10        if $S.Q.size > 1$ and
          $S.Q.head.time + S.Q.head.amt \geq S.Q.head.next.time$ then
11           $a \leftarrow S.Q.head.amt$
12           $b \leftarrow S.Q.head.time$
13           $S.Q.pop$   ▷ remove head from queue
14           $S.Q.head.amt \leftarrow S.Q.head.amt + a$
15           $S.Q.head.time \leftarrow b$
16     if $S.capacity = 0$ then  ▷ $S.Q.head.time > Now$
17        $S.priority \leftarrow S.background\_priority$
18        if $\neg S.is\_blocked$ then
19           $S.replenishment.enqueue(S.Q.head.time)$
```

Figure 6.3: Pseudo-code for budget over-run check.

Recall that amplification occurs when a replenishment is consumed in its entirety with some amount of overrun. This overrun amount is added to the replenishment and scheduled at a time in the future. The POSIX sporadic server is never charged for the overrun time because a negative budget is immediately set to zero.

A simple fix would be to just allow the currently available execution capacity to become negative. This prevents the amplification effect by keeping the capacity plus replenishments equal to the initial budget, thereby limiting execution within a server period to at most the initial budget plus the maximum overrun amount. However, since the replenishment is larger by the overrun amount,

```
SPLIT_CHECK

1  if $S.usage > 0$ and $S.Q.head.time \leq Now$ then
2      $remnant \leftarrow S.Q.head.amt - S.Usage$
3      ▷ R is a new replenishment data structure
4      $R.time \leftarrow S.Q.head.time$
5      $R.amt \leftarrow S.usage$
6      if $S.Q.size = S.max\_Repl$ then
           ▷ Merge remnant with next replenishment
7          $S.Q.pop$
8          if $S.Q.size > 0$ then
9              $S.Q.head.amt \leftarrow S.Q.head.amt + remnant$
10         else
11             $R.amt \leftarrow R.amt + remnant$
12     else
           ▷ Leave remnant as reduced replenishment
13         $S.Q.head.amt \leftarrow remnant$
14         $S.Q.head.time \leftarrow S.Q.head.time + S.usage$
       ▷ Schedule replenishment for the consumed time
15     $R.time \leftarrow R.time + S.period$
16     $S.Q.add(R)$
17     $S.usage \leftarrow 0$
```

Figure 6.4: Pseudo-code for conditionally splitting a replenishment.

the server is still not being properly charged for the overrun. The resulting interference is therefore greater by the overrun amount for each period.

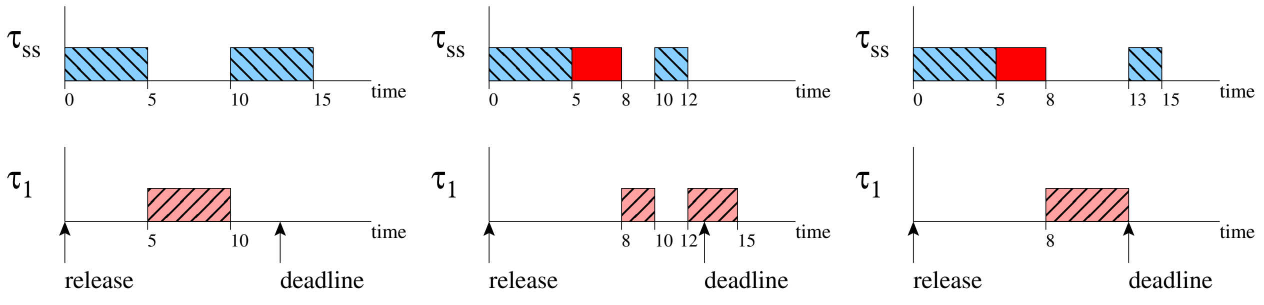
A more effective limit on the resulting interference due to overruns can be achieved by charging it against a future replenishment. To be clear, this is not preventing the overrun from occurring, only once it does occur, future overruns of the same size will be prevented from accumulating and only permitted to occur once per interval of continuous server execution. It should be noted that since the overrun can occur, it must be taken into account in the schedulability analysis.

In the BUDGET_CHECK procedure, the compensation for an overrun is handled. An overrun has occurred when the sum of the replenishment amounts with times less than or equal to the current time exceed the server's current usage (*S.usage*). This overrun amount is charged against future replenishments as if that time has already been used. The while loop starting at line 2 of BUDGET_CHECK iterates through the replenishments, charging as needed until the *S.usage* is less

```
Unblock_Check
      ▷ Advance earliest activation time to now.
 1    if $S.capacity > 0$ then
 2        if $S.priority \neq S.foreground\_priority$ then
 3            $S.priority \leftarrow S.foreground\_priority$
 4        $S.Q.head.time \leftarrow Now$
          ▷ Merge available replenishments
 5        while $S.Q.Size > 1$ do
 6            $a \leftarrow S.Q.head.amt$
 7            if $S.Q.head.next.time \leq Now + a - S.Usage$ then
 8                $S.Q.pop$ ▷ remove head from queue
 9                $S.Q.head.amt \leftarrow S.Q.head.amt + a$
10                $S.Q.head.time \leftarrow Now$
11            else
12                exit
13    else
14        $S.replenishment.enqueue(S.Q.head.time)$
```

Figure 6.5: Pseudo-code for unblocking event.

than the replenishment at the head of the *S.Q*. At this point, the overrun amount will remain in *S.usage*. Therefore, when the next replenishment arrives it will immediately be reduced by the amount in *S.usage* according to the calculation of *S.capacity*. This prevents the POSIX amplification effect by ensuring that overrun amounts are considered as borrowing from future replenishments.



(a) normal operation (b) overrun affecting deadline of τ_1 (c) postponing replenishment time

Figure 6.6: Postponing replenishments after an overrun.

The intervals of sporadic server execution that correspond to a given replenishment are spaced apart by the server's period. This spacing allows lower priority tasks to receive execution time.

However, when there is an overrun, the time between such intervals may shrink. For instance, consider Figure 6.6a. The execution of τ_1 takes place between the two intervals of server execution. However, if an overrun occurs in the first execution interval of τ_{ss}, the time meant for τ_1 is occupied, forcing τ_1 to start execution later. Since the next replenishment of τ_{ss} arrives during the execution of τ_1, τ_{ss} is permitted to preempt τ_1. This further postpones the completion of τ_1, which may cause a missed deadline as illustrated in Figure 6.6b.

The overrun amount is considered as time borrowed from future replenishments and logically, the borrowed time should be taken from the earliest available replenishment time. Only subtracting a portion of a replenishment without modifying its activation time would be effectively borrowing the latest available portion of the replenishment time rather than the earliest. To borrow from the earliest portion of a replenishment, the activation time is increased by the borrowed amount. This is done in line 9 of the BUDGET_CHECK procedure. By borrowing from the earliest portion of a replenishment, Figure 6.6c illustrates that the example using τ_1 is able to meet its deadline.

6.5.2 Correcting the Premature Replenishments

Premature replenishments occur when one or more unit-capacity servers violate the *sporadic constraint*. The POSIX sporadic server experiences premature replenishments due to its simplified tracking of activation times. When a replenishment arrives, it is immediately merged with any replenishment that has an activation time less than or equal to the current time. This may result in invalid merging of replenishment cohorts, allowing a replenishment to be used earlier than the earliest activation time that would be consistent with the sporadic constraint.

To maintain the *sporadic constraint*, each cohort must be separated by at least the server's period (T_s). In the corrected algorithm, replenishments can be totally or partially consumed. If the entire replenishment is consumed, the replenishment time is set to $R.time + S.Period$ (line 6 of BUDGET_CHECK). When only a portion of time is used, the replenishment must be split resulting in distinct replenishments. This is performed in the SPLIT_CHECK procedure. Only the used portion is given a new time, T_s in the future (line 8). As these replenishments are maintained distinct and each usage is separated by at least T_s, the worst-case interference is correctly limited in the fashion consistent with the ideal model.

The number of replenishments to maintain over time can become very large. To help minimize this fragmentation the corrected algorithm allows merging of replenishments in accordance with

Observation 3. This is achieved through lines 11–13 of the Budget_Check procedure. If the replenishment at the head of the queue overlaps with the next replenishment in the queue, the two will be merged.

posix limits the number of replenishments into which the server capacity can be fragmented. The parameter *sched_ ss_ max_ repl* defines the maximum number of replenishments that can be pending at any given time. There are pragmatic reasons for this limit. One is that it allows pre-allocation of memory resources required to keep track of replenishments. Another is that it implicitly bounds the number of timer interrupts and context switches that replenishments can cause within the server period. This effect can be translated into the ideal model using Observation 2 as follows: When the pending replenishment limit is reached all server units with earliest-next-activation time prior to the next replenishment time are advanced to the next replenishment time, effectively becoming part of the next replenishment/cohort. This action is performed in the else block starting at line 6, of the Split_Check procedure.

6.5.3 Improving Temporal Isolation

As explained in Section 6.4.3, the posix standard currently permits an interpretation that the sporadic server's background priority range is limited, and so there is no mechanism to guarantee that a sporadic server will not starve tasks that are scheduled with policies below the real-time range.

To permit all other tasks on the system to be isolated from budget overruns of a sporadic server, the proposed design is that the allowable range of priorities for a server's background priority extend down to include the lowest system priority, and further include an extreme value so low that a thread with that value can never run. Alternatively one could introduce a new scheduling parameter or configurable system parameter to indicate that instead of switching to a background priority when a sporadic server is out of budget, it should wait until its next replenishment time. Adding either of these features would require very little change to an existing, implemented scheduler, and would provide at least one portable way to write applications with temporal isolation. If introducing such a new priority value or a new interface exceeds the latitude of the ieee Standards interpretation process, the next best thing is to make it explicit that implementations are permitted to define the range of priorities for sched_sporadic to extend below that of sched_fifo.

To demonstrate the usefulness of priority ranges we extended our simulator to provide such functionality. This allowed us to implement SCHED_SPORADIC alongside a variety of other scheduling policies, including earliest-deadline-first (EDF) and deadline sporadic. We used a single range of 64-bit integer values to cover both static priorities and deadlines, reserving priority ranges at the extreme low and extreme high ends, and interpreting the largest expressible value as "do not execute". Of course, such an implementation model needs remapping of values at the API in order to comply with POSIX, which interprets numerically large values as higher fixed priorities, and requires a contiguous range of values for each policy.

6.5.4 Evaluation

This section presents our evaluation of the problems and the proposed solutions discussed above. This evaluation was performed using an implementation in the Linux-2.6.28 operating system and an in-house simulator.

It is perhaps an indication of the seriousness of the budget amplification effect that we discovered it accidentally, as users. We were experimenting with a version of the Linux kernel that we had modified to support the SCHED_SPORADIC policy. The reason for using this policy was to bound the scheduling interference device driver threads cause other tasks [61]. Our implementation appeared to achieve the desired effect. We noticed that the server was consuming more execution time than its budget. We attributed these overruns to the coarseness of our sporadic server implementation, which enforced budget limits in whole ticks of the system clock. Since the clock tick was much larger than the network message inter-arrival and processing times, this allowed the execution behavior of the sporadic server under network device driver loads to be very similar to that of a periodic server, and it was able to run over significantly in each period. We hoped that by going to a finer-grained timer, we could both reduce the overrun effect and distinguish better between the sporadic and periodic servers. Therefore, we tried using a different Linux implementation of the sporadic server, developed by Dario Faggioli [29], which uses high-resolution timers. With a few refinements, we were able to repeat our prior experiments using this version, but the server continued to run significantly over its budget – sometimes nearly double its allocated CPU bandwidth. After first checking for errors in the time accounting, we analyzed the behavior again, and conjectured that the overruns might be due to budget amplification. To test this conjecture, we modified the scheduler to allow the currently available execution capacity to become negative, and observed the server CPU utilization drop down to the proper range.

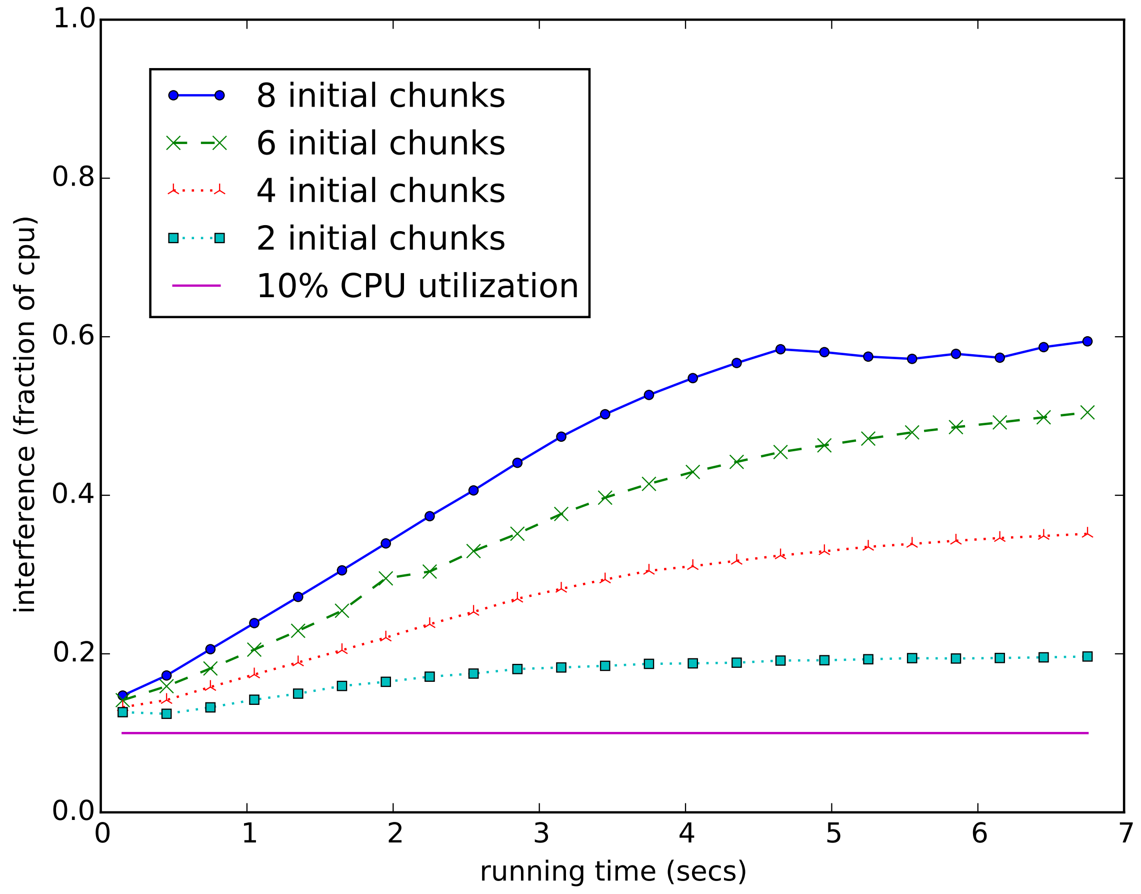


Figure 6.7: Budget amplification effect with varying number initial replenishments (empirical measurement).

As further verification, we conducted a simple structured experiment, using the Linux sporadic server implementation. A sporadic server is given a period of 10 msecs and a budget of 1 msec. Two jobs arrive, with execution times of one-half the budget and one-half the server period. The effect is to divide the budget into two replenishments. Immediately following the second job arrival, more jobs arrive, with the same execution times as the initial jobs, at a rate that maintains a server backlog for the duration of the experiment. The results are seen in the lower trace of Figure 6.7. Each replenishment, originally one-half the budget, is able to increase to the size of the full budget, allowing the server to achieve double its budgeted CPU utilization. The other traces in Figure 6.7 show what happens if the number of active replenishments before the start of the overload is 4, 6, and 8. In principle, with sufficiently many initial fragments before the overload interval, the server CPU utilization could reach nearly 100%. However, in our experiment, the increase in server utilization did not climb so quickly, apparently due to the replenishments overlapping, causing merging of replenishments.

To further understand the sporadic server anomalies, a simulator was developed. With the simulator we were able to reduce the scheduling “noise" allowing us to focus on the problems associated with the POSIX definition of sporadic server rather than those introduced by the hardware and Linux kernel. Figures 6.8 and 6.9 are from this simulator.

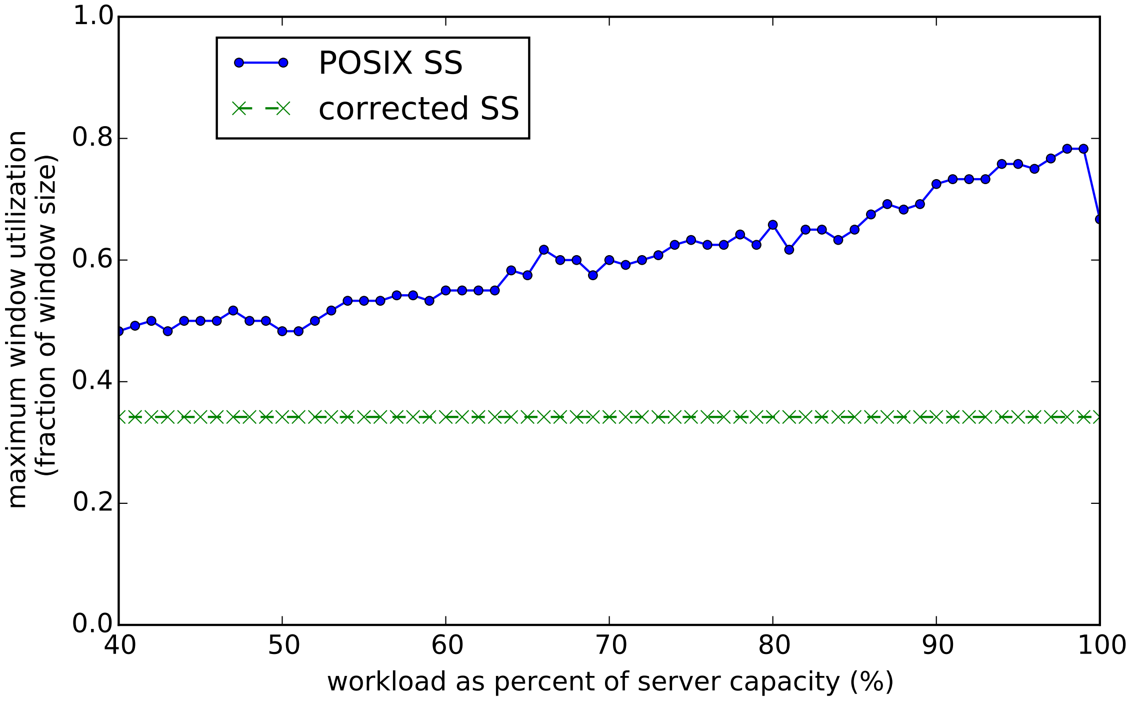


Figure 6.8: Maximum measured utilization in a time window of 120 time units for a randomly generated workload served by a sporadic server (budget=40, period=120) (simulation study).

The effects of budget amplification can not only increase the total utilization over an entire run, but also the maximum demand for execution time in any time window of a given size.[2] A correctly operating sporadic server should have the same worst-case demand as an equivalent periodic task.

So, if we consider a period-sized time window, the maximum server demand in that window should not exceed the execution time divided by the period.

[2]Here “demand” is considered to be the amount of processor time for which the server is allowed to compete at its foreground priority.

Due to the budget amplification effect, the expected limit on utilization is not true for the POSIX sporadic server. Figure 6.8 shows the amount of execution time achieved by sporadic server at its native priority (here the sporadic server is not allowed to run at background priority). This experiment was performed using an exponential distribution of job execution times with a mean job execution time of 10 time units. The server's period is 120 and the budget is 40. To demonstrate the budget amplification, there must be overruns. Here each job is permitted to overrun 1 time unit, corresponding to the resolution of the execution time clock as defined in the POSIX sporadic server. The inter-arrival times of jobs are also determined with an exponential distribution where the mean arrival rate is adjusted to create an average workload as a percent of server capacity. So, for a workload of 100% the mean inter-arrival time would be $\frac{120}{4} = 30$. The corrected sporadic server provides the expected maximum of 34% utilization in a given window $\left(\frac{40+1}{120}\right)$. The POSIX implementation, however, drastically exceeds the maximum utilization, clearly not providing the expected temporal isolation. (One may notice that the maximum window utilization drops slightly near 100% server capacity and is likely due to more frequent overlapping and merging of replenishments.)

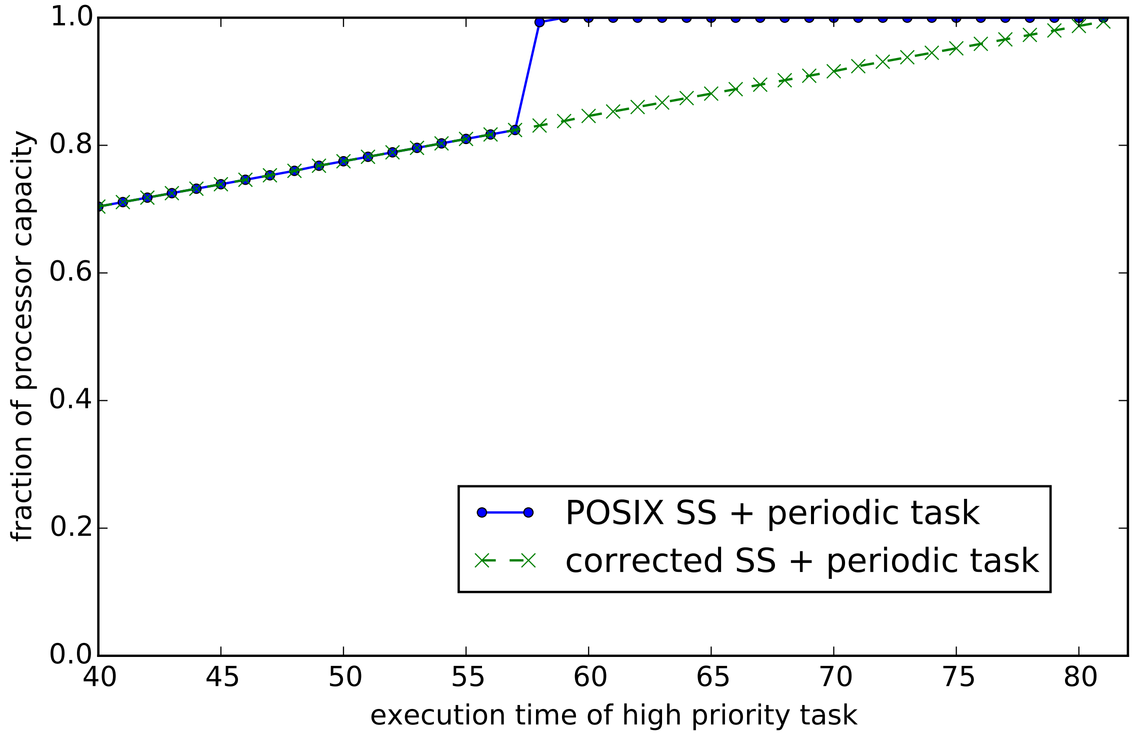


Figure 6.9: Effect of premature replenishments (simulation study).

To demonstrate the effect of premature replenishments Figure 6.9 graphs the measured combined capacity of the sporadic server and a higher priority periodic task. The periodic task has a period of 141 and execution time identified by the value along the x-axis. The sporadic server has a budget of 42 and a period of 100. The effect of the premature replenishment is not seen until the execution time of the high priority periodic task increases above 57 time units.

At this point the effect hits abruptly, and the POSIX sporadic server is able to acquire 58 percent of the CPU. This is an increase of 38 percent $\left(\frac{.16}{.42}\right)$ from its budgeted 42 percent maximum and causes the CPU to become saturated. The corrected sporadic server is able to correctly limit the CPU utilization, thereby allowing other tasks to run, despite the server overload.

Attempts were made to demonstrate this premature replenishment effect on random arrivals and execution times, however, it appears that the effect does not occur often enough to be measured on a macroscopic scale. If, as this suggests, the premature replenishment anomaly has a very low probability, it may be that this anomaly would only be a concern in a hard real-time environment.

6.6 Overhead and Efficiency

To evaluate the response time characteristics of our sporadic server, we measured the response time of datagram packets sent across a network. The response time of a packet is measured by the time difference between sending the packet on one machine, m_1, and receiving the packet by another, m_2. More specifically, the data portion of each packet sent from m_1 contains a timestamp, which is then subtracted from the time the packet is received by the UDP layer on m_2.[3] In our setup, m_1 periodically sends packets to m_2. The time between sending packets is varied in order to increase the load experienced by the network receive thread on m_2. The receive thread on m_2 is scheduled using either the polling server, sporadic server, or *SCHED_FIFO* [43] scheduling policies.[4] In our experiments, m_2 is running Linux 2.6.38 with a ported version of softirq threading found in the 2.6.33 Linux real-time patch. m_2 has a Pentium D 830 processor running at 3GHz with a 2x16KB L1 cache and a 2x1MB L2 cache. 2GB of RAM are installed. The kernel was configured to use only one core, so all data gathered is basically equivalent to a uniprocessor system.

[3]The clocks for the timestamps on m_1 and m_2 are specially synchronized using a dedicated serial connection.

[4]*SCHED_FIFO* differs from the other in allowing thread of sufficiently high priority to execute arbitrarily long without preemption.

Scheduling the Linux network receive thread (i.e., *sirq-net-rx*) using various scheduling policies affects the average response time of received network packets. One would expect that the polling server would result in higher average response times than *SCHED_ FIFO* or sporadic server and that sporadic server and *SCHED_ FIFO* should provide similar average response times until sporadic server runs out of budget.

In our experiment, sporadic server and polling server are both given a budget of 1 millisecond and a period equal to 10 milliseconds. The sporadic server's maximum number of replenishments is set to 100. The hourglass task is scheduled using *SCHED_ FIFO* scheduling at a real-time priority lower than the priority of the network receive thread. Each data point is averaged over a 10 second interval of sending packets at varied rates. The CPU utilization and response time for the described experiment are shown in Figures 6.10 and 6.11.

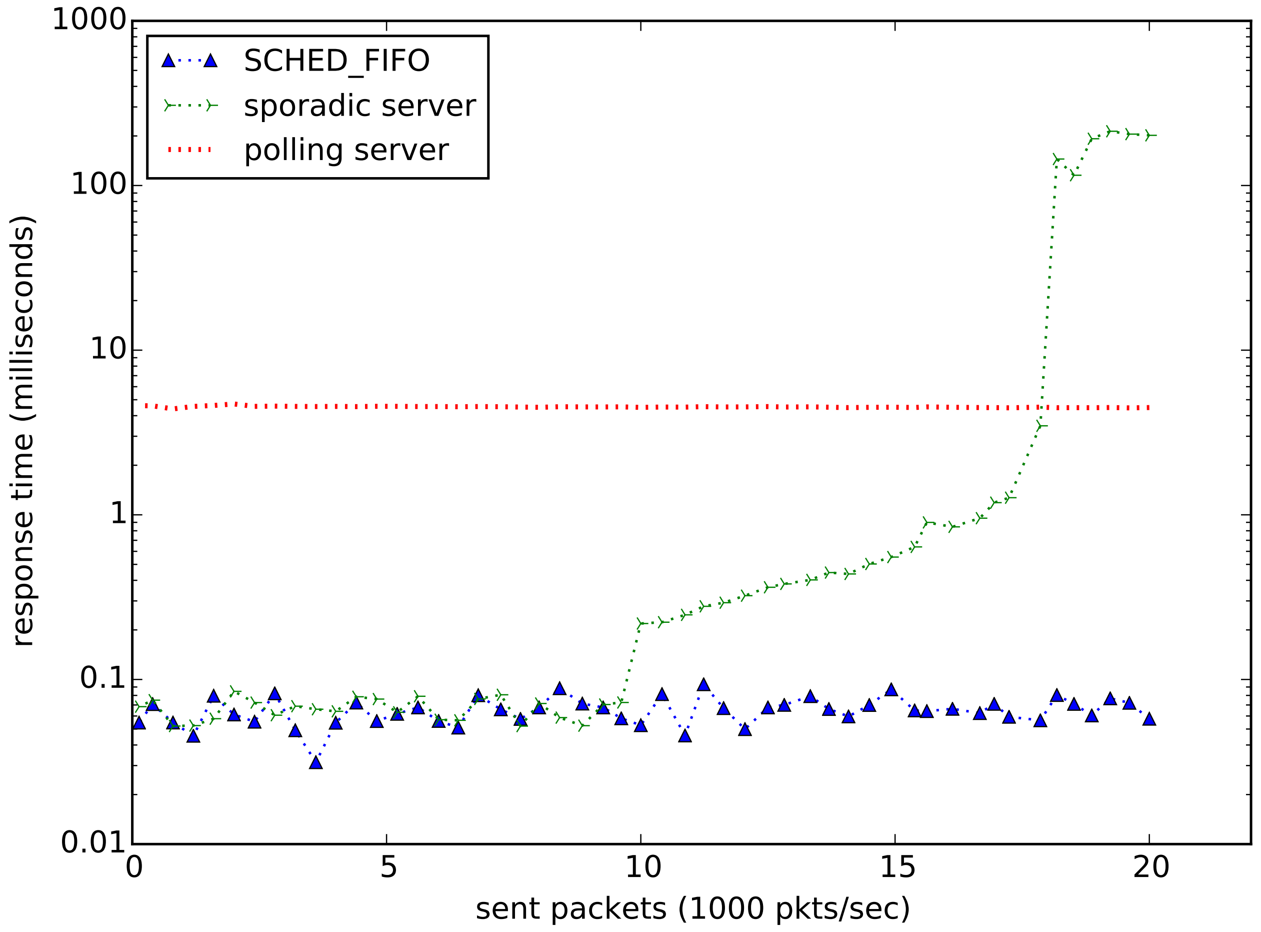


Figure 6.10: Response time using different scheduling policies.

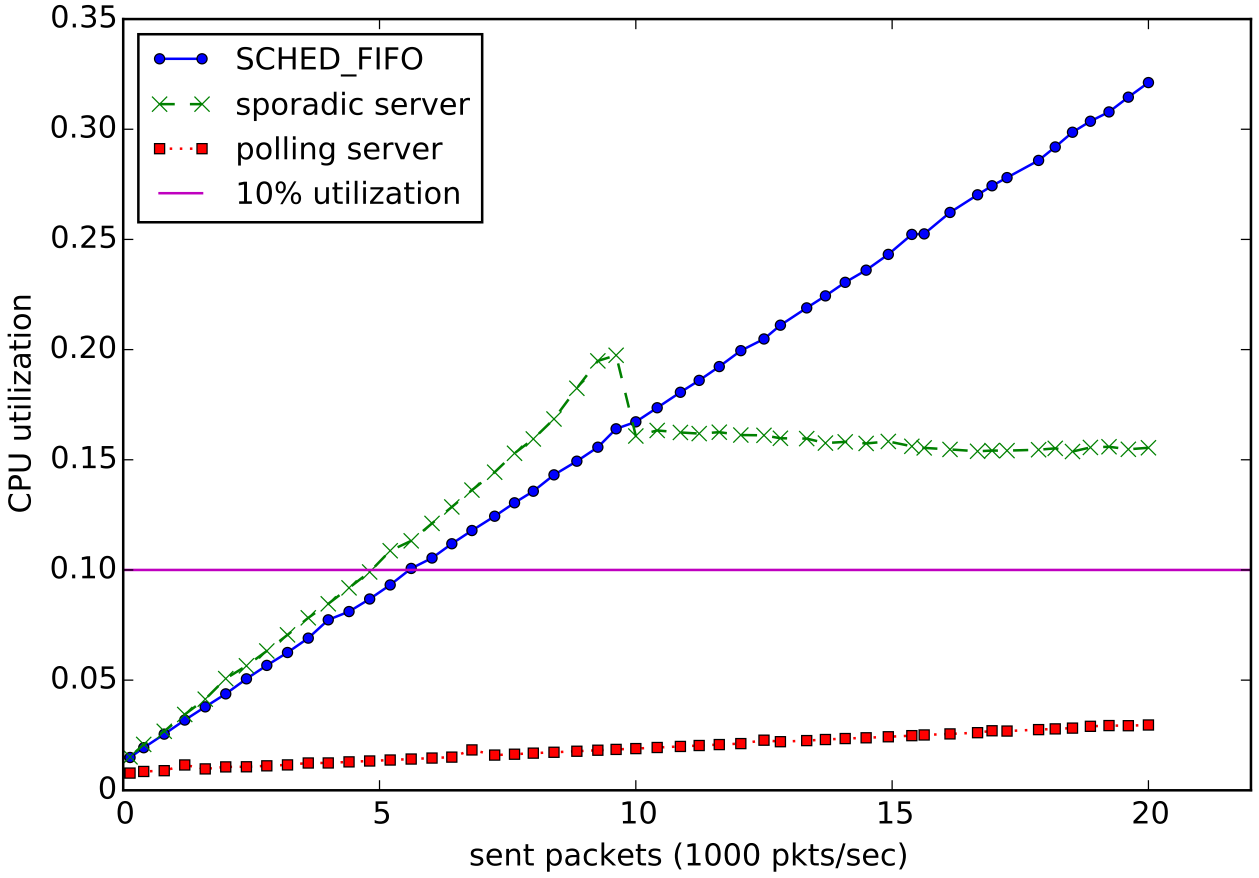


Figure 6.11: sirq-net-rx thread CPU utilization using different scheduling policies.

One would expect that if the sporadic server and polling server both were budgeted 10% of the CPU, the lower-priority hourglass task should be able to consume at least 90% of the CPU time regardless of the load. However, the data for the experiment shows that the sporadic server is causing much greater than 10% interference. The additional interference is the consequence of preemptions caused by the server. Each time a packet arrives the sporadic server preempts the hourglass task, thereby causing two context switches for each packet arrival. Given that the processing time for a packet is small (2-10 microseconds) the server will suspend itself before the next packet arrives. In this situation, the aggregate time for context switching and other sporadic server overhead such as using additional timer events and running the sporadic-sever-related accounting becomes significant. For instance, on the receiving machine the context-switch time alone was measured at 5-6 microseconds using the *lat_ ctx* LMbench program [76].

The overhead associated with preemption causes the additional interference that is measured by the lower-priority hourglass task.[5]

A snapshot of CPU execution time over a 500 microsecond time interval was produced using the Linux Trace Toolkit (LTTng) [87] and is shown in Figure 6.12. The top bar is the *sirq-net-rx* thread and the bottom bar is the lower-priority hourglass measuring task. This figure shows that the CPU time of both tasks is being finely sliced. The small time slices cause interference for both the lower-priority and sporadic server thread that would not be experienced if the threads were able to run to completion.

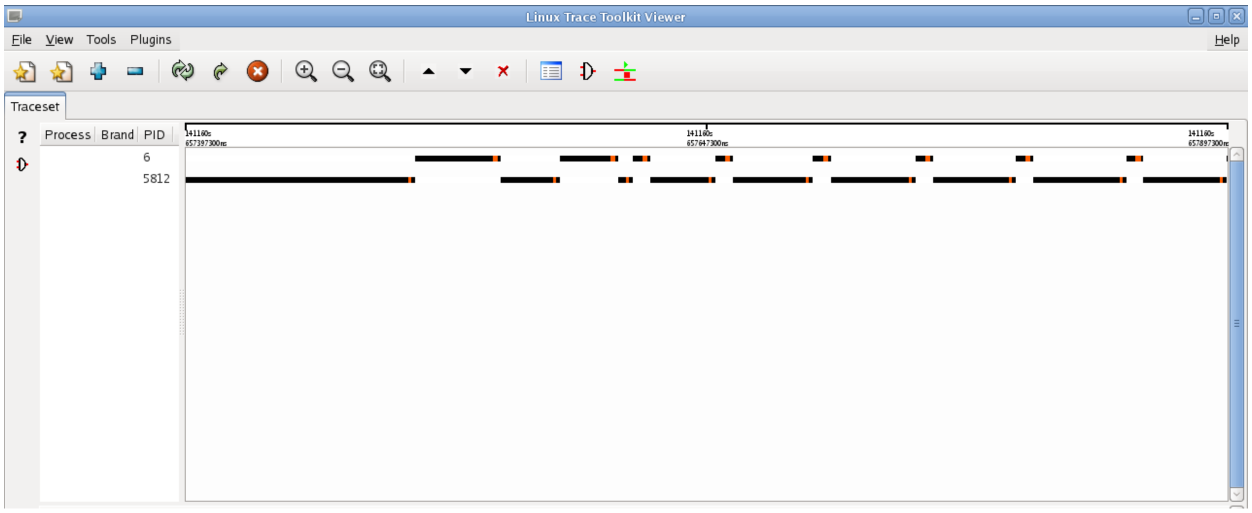


Figure 6.12: LTTng visualization of CPU execution.

6.6.1 Accounting for Preemption Overhead

To ensure that no hard deadlines are missed, and even to ensure that soft deadlines are met within the desired tolerances, CPU time interference due to preemptions must be included in the system's schedulability analysis. The preemption interference caused by a periodic task can be included in the analysis by adding a preemption term to the task's worst-case execution time ($WCET$) that is equal to twice the worst-case context-switch cost (CS_{time}) – one for switching into the task and

[5]The lower-priority thread does not measure much of the cache eviction and reloading that other applications may experience, because its code is very small and typically remains in the CPU's cache. When cache effects are taken into account, the potential interference penalty for each preemption by a server is even larger.

one for switching out of the task.[6] Assuming all tasks on the system are periodic, this is at least a coarse way of including context-switch time in the schedulability analysis.

A sporadic server can cause many more context switches than a periodic task with the same parameters. Rather than always running to completion, a sporadic server has the ability to self-suspend its execution. Therefore, to obtain a safe $WCET$ bound for analysis of interference,[7] one would have to determine the maximum number of contiguous "chunks" of CPU time the sporadic server could request within any given period-sized time interval. The definition of sporadic- server scheduling given in scheduling theory publications does not place any such restriction on the number of CPU demand chunks and thus imposes no real bound on the $WCET$. In order to bound the number of preemptions, and thereby bound the time spent context switching, most implemented variations of sporadic server limit the maximum number of pending replenishments, denoted by max_repl. Once max_repl replenishments are pending, a sporadic server will be prevented from executing until one of the future replenishments arrives. Using max_repl, the maximum number of context-switches per period of a sporadic server is two times the max_repl. Using this logic, and assuming that the actual context-switch costs are added on top of the servers budget, a worst-case upper bound on the interference that can be caused by a sporadic server task could be written as:

$$SS_{budget} + (2 * max\_repl * CS_{time})$$

Accounting for the cost due to preemptions is important in order to ensure system schedulability; however, adding preemption cost on top of the server's budget as above results in over-provisioning. That is, if a sporadic server does not use max_repl number of replenishments in a given period a worst-case interference bound derived in this way is an over-estimate. At the extreme, when a sporadic server consumes CPU time equal to its budget in one continuous chunk, the interference only includes the cost for two context switches rather than two times max_repl. However, the

[6]This is an intentional simplification. The preemption term should include all interferences caused by the sporadic server preempting another thread, not only the direct context-switch time, but also interferences such as the worst-case penalty imposed by cache eviction and reloading following the switch. For checking the deadline of a task, both "to" and "from" context switches need to be included for potentially preempting task, but only the "to" switch needs be included for the task itself.

[7]From this point on we abuse the term $WCET$ to stand for the maximum interference that a task can cause for lower-priority tasks, which includes not just the maximum time that the task itself can execute, but also indirect costs, such as preemption overheads.

server cannot make use of this windfall to execute jobs in its queue because the context switch cost was not added to its actual budget.

We believe a better approach is to account for actual context-switch costs while the server is executing, charging context switch costs caused by the server against its actual budget, and doing so only when it actually preempts another task. In this approach the SS_{budget} alone is used as the interference bound for lower-priority tasks. Accounting for context-switching overhead is performed on-line by deducting an estimate of the preemption cost from the server's budget whenever the server causes a preemption. Charging the sporadic server for preemption overhead on-line reduces over-provisioning, and need not hurt server performance on the average, although it can reduce the effective worst-case throughput of the server if the workload arrives as many tiny jobs (as in our packet service experiment).

Charging for preemptions on-line requires that the preemption interference be known. Determining an appropriate amount to charge the server for preempting can be very difficult, as it depends on many factors. In order to determine an amount to charge sporadic server for a preemption, we ran the network processing experiment under a very heavy load and extracted an amount that consistently bounded the interference of sporadic server to under 10%. While such empirical estimation may not be the ideal way to determine the preemption interference, it gave us a reasonable value to verify that charging for preemptions can bound the interference.

The network experiment was performed again, this time charging sporadic server a toll of 10 microseconds each time it caused a preemption. Figure 6.13 shows the results for the experiment and demonstrates that time interference for other lower-priority tasks can be bounded to 10%, that is, the server's budget divided by its period.

6.6.2 Preemption Overhead

Bounding the interference that an aperiodic workload causes for other tasks is the primary objective of aperiodic server scheduling; however, one would also like to see fast average response time. Figure 6.14 shows that under heavy load, the average response time of packets when using sporadic-server scheduling is actually worse than that of a polling server with the same parameters. For this experiment, not only is the sporadic server's average response time higher, but as the load increases up to 45% of the packets were dropped.[8]

[8]No packets were dropped by the other servers.

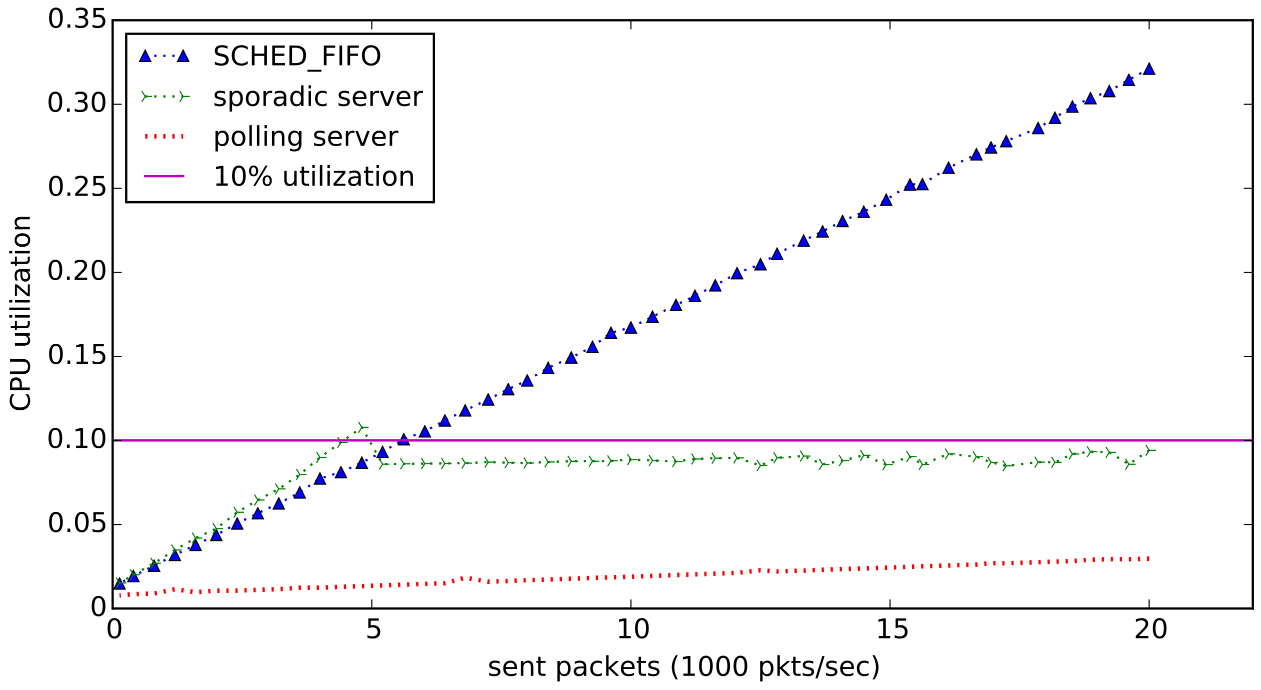


Figure 6.13: Properly bounding CPU utilization by accounting for context-switching overhead.

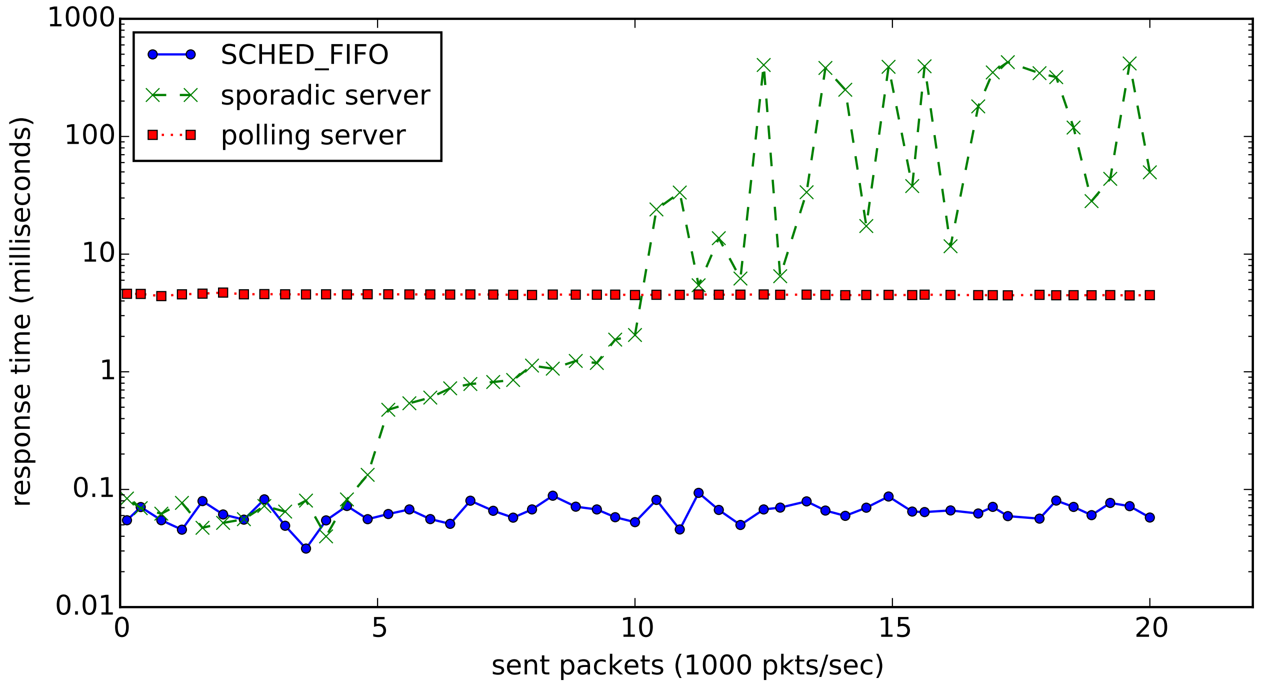


Figure 6.14: Effect on response time when properly bounding CPU utilization.

The poor performance of the sporadic server is due to a significant portion of its budget being consumed to account for preemption costs, leaving a smaller budget to process packets. If all of the packets arrived at the same time, the processing would be batched and context switching would not occur nearly as often. However, due to the spacing between packet arrivals, a large number of preemptions occur. A polling server on the other hand has a much larger portion of its budget applied to processing packets, and therefore does not drop packets and also decreases the average response time.

Based on the poor performance of a sporadic server on such workloads one might naïvely jump to the conclusion that, in general, a polling server is a much better choice. Actually, there is a trade-off, in which each form of scheduling has its advantage. Given the same budget and period, a sporadic server will provide much better average-case response time under light load, or even under a moderate load of large jobs, but can perform worse than the polling server for certain kinds of heavy or bursty workloads.

It turns out that the workload presented by our packet service example is a poor one for the sporadic server, in that a burst of packet arrivals can fragment the server budget, and then this fragmentation becomes "locked in" until the backlog is worked off. Suppose a burst of packets arrives, and the first *max_repl* packets are separated by just enough time for the server to preempt the running task, forward the packet to the protocol stack, and resume the preempted task. The server's budget is fragmented into *max_repl* tiny chunks. Subsequent packets are buffered (or missed, if the device's buffer overflows), until the server's period passes and the replenishments are added back to its budget. Since there is by now a large backlog of work, the server uses up each of its replenishment chunks as it comes due, then suspends itself until the next chunk comes due. This results in a repetition of the same pattern until the backlog caused by the burst of packets has been worked off. During this overload period, the sporadic server is wasting a large fraction of its budget in preemption overhead, reducing its effective bandwidth below that of a polling server with the same budget and period. There is no corresponding improvement in average response time, since after the initial *max_repl* fragmentation, the reduced bandwidth will cause the response times to get worse and worse.

6.6.3 Reducing the Impact of Preemption Overhead

A hybrid server combining the strengths of polling and sporadic servers may be a better alternative than choosing either one. In this approach, a sporadic server is used to serve light loads and a polling server to serve heavy loads.

Sporadic-server scheduling supports a polling-like mode of operation. By simply setting the *max_repl* parameter value to one, only a single preemption is permitted in any time interval equal to the server's period.

When changing modes of operation of the sporadic server in the direction of reducing *max_repl*, something must be done if the current number of pending replenishments would exceed *max_repl*. One approach is to allow the number of pending replenishments to exceed *max_repl* temporarily, reducing it by one each time a replenishment comes due. Another approach is to implement the reduction at once, by coalescing pending replenishments. This is similar to the classical mode-change scheduling problem, in that one must be careful not to violate the assumptions of the schedulability analysis during the transition. In the case of a sporadic server the constraint is that the server cannot cause any more interference within any time window than would be caused by a periodic task with execution time equal the server budget and period equal to the server's budget replenishment period, including whatever adjustments have been made to the model to allow for context-switch effects. We call this the *sliding window constraint* for short.

In order to maintain the sliding-window constraint during the mode change, one can think in terms of changing the times associated with pending replenishments. Consolidating the replenishment times would allow the creation of a single replenishment with an amount equal to the server's initial budget. To guard against violating the sliding-window constraint, the replenishment time of any replenishment must not be moved earlier in time. One approach is to coalesce all replenishments into the replenishment with a time furthest in the future, resulting into a single replenishment with an amount equal to the server's initial budget as shown in Figures 6.15 and 6.16.

Switching from sporadic server to a polling-like server should be performed if the server is experiencing heavy load. The ideal switching point may be difficult to detect. For instance, a short burst may be incorrectly identified as the onset of a heavy load and the early switching may cause the server to postpone a portion of its budget that could have been used sooner. Conversely, delaying the switch may mean that time that could have been used to serve incoming jobs is wasted on preemption charges.

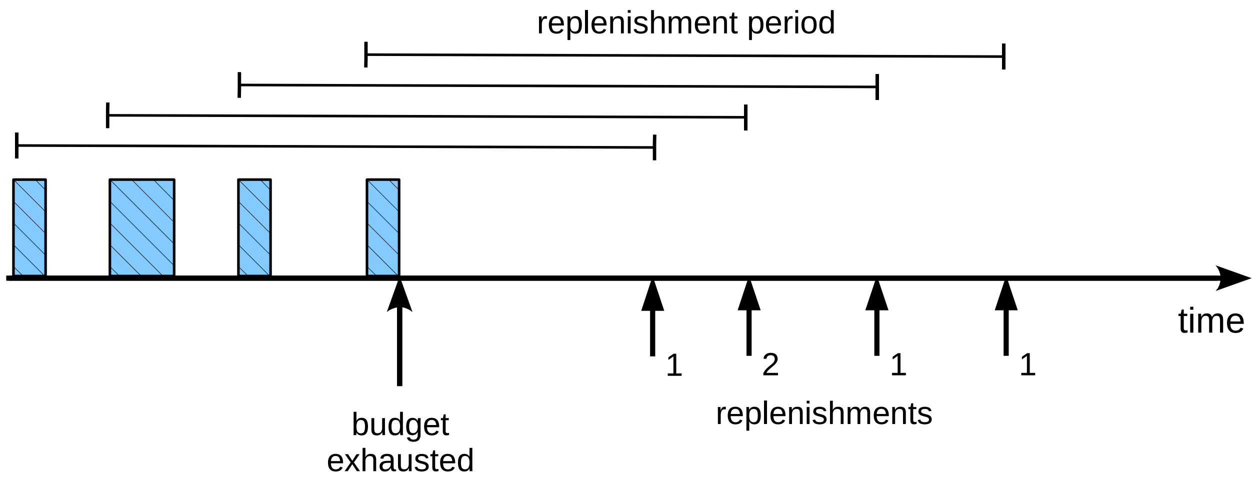


Figure 6.15: Sporadic server with $max\_repl \geq 4$, before switch to polling-like server.

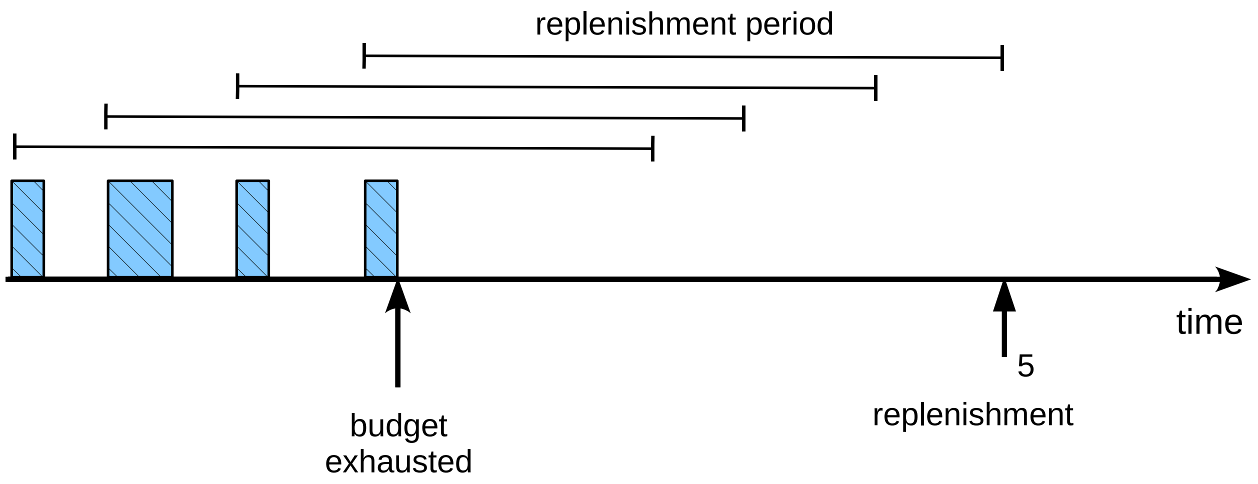


Figure 6.16: After switch to poll-like server, with $max\_repl = 1$ and replenishments coalesced.

While an ideal switching point may not be possible to detect beforehand, one reasonable indicator of a heavy load is when sporadic server uses all of its budget. That is the point when a sporadic server is blocked from competing for CPU time at its scheduling priority. At this point the server could switch to its polling-like mode of operation.

A possible event to indicate when to switch back to the sporadic server mode of operation is when a sporadic server blocks but still has available budget. This point in time would be considered as entering a period of light load and the *max_repl* could be reinstated.

Implementation of the switching mechanism described above is relatively simple. The replenishments are coalesced into a single replenishment when the server runs out of budget but still has work. The single replenishment limit will remain enforced until the sporadic server is suspended and has budget, a point in time considered to be an indication of light load. So, the polling-like mode of operation will naturally transition back to the original sporadic server mode of operation.

Immediately coalescing all replenishments may be too eager. Loads that are between light and heavy may experience occasional or slight overloads that require only slightly more CPU time. In this case, converting all potential preemption charges, by delaying replenishments, into CPU time to serve packets is too extreme. Therefore, to perform better under a range of loads one approach is to coalesce only two replenishments for each overload detection. Using this method allows the sporadic server to naturally find an intermediate number of replenishments to serve packets efficiently without wasting large portions of its budget on preemption charges.

The performance data for the two coalescing methods, immediate and gradual, are shown in Figures 6.17 and 6.18. These figures show the advantage of transitioning between sporadic-server and polling-like mode of operation. Between light load and approximately 4500 pkts/sec, the sporadic server has the response times closely matching *SCHED_FIFO* scheduling. However, once the load is heavy enough the sporadic server is forced to limit the amount of CPU demand and therefore the response time begins to increase to that of a polling server. There is not enough CPU budget to maintain the low average response time with *SCHED_FIFO* scheduling. The difference between the immediate and gradual coalescing is seen when the restriction on CPU demand begins. The gradual coalescing provides a gradual transition to polling-like behavior whereas the immediate coalescing has a much faster transition to the polling server's response time performance. The better performance of the gradual coalescing is due to the server making better use of the available budget. With immediate coalescing, when the server transitions to the polling-like mode the CPU utilization drops, as one would expect of sporadic server where the *max_repl* is set to 1. However, with gradual coalescing the server continues to use its available budget to pay for preemption costs and serve some jobs earlier, which results in lower response times.

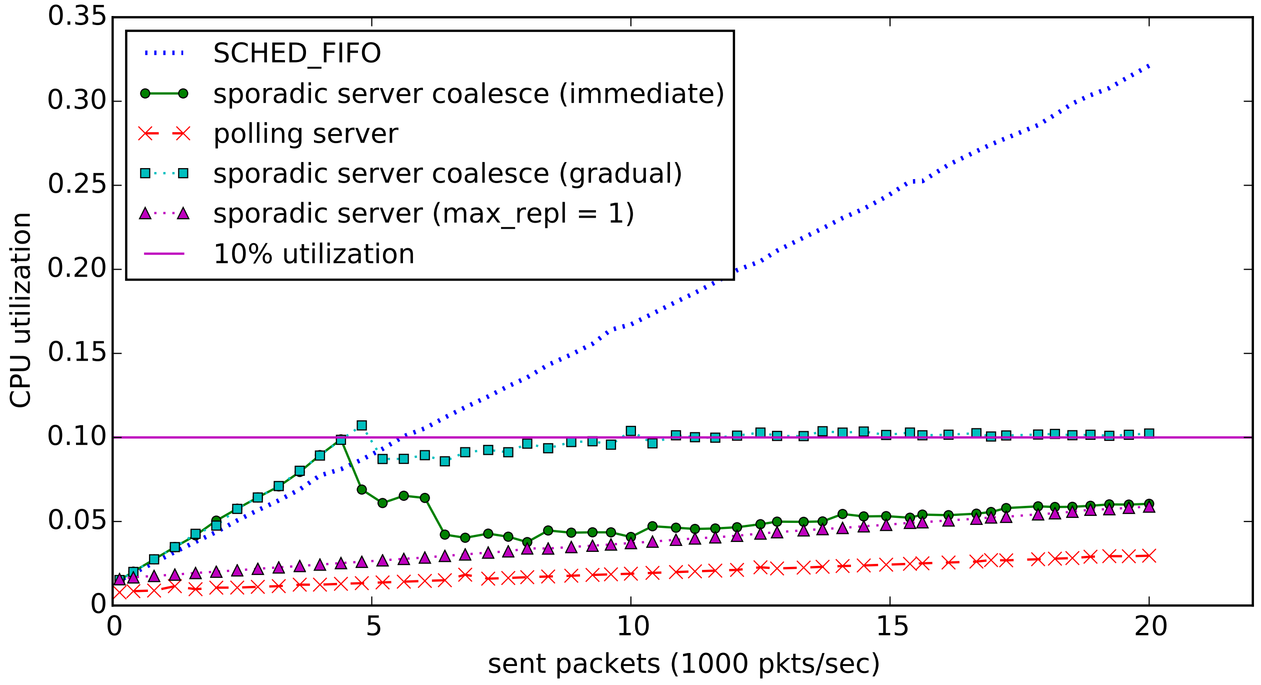


Figure 6.17: CPU utilization when coalescing replenishments under heavy load.

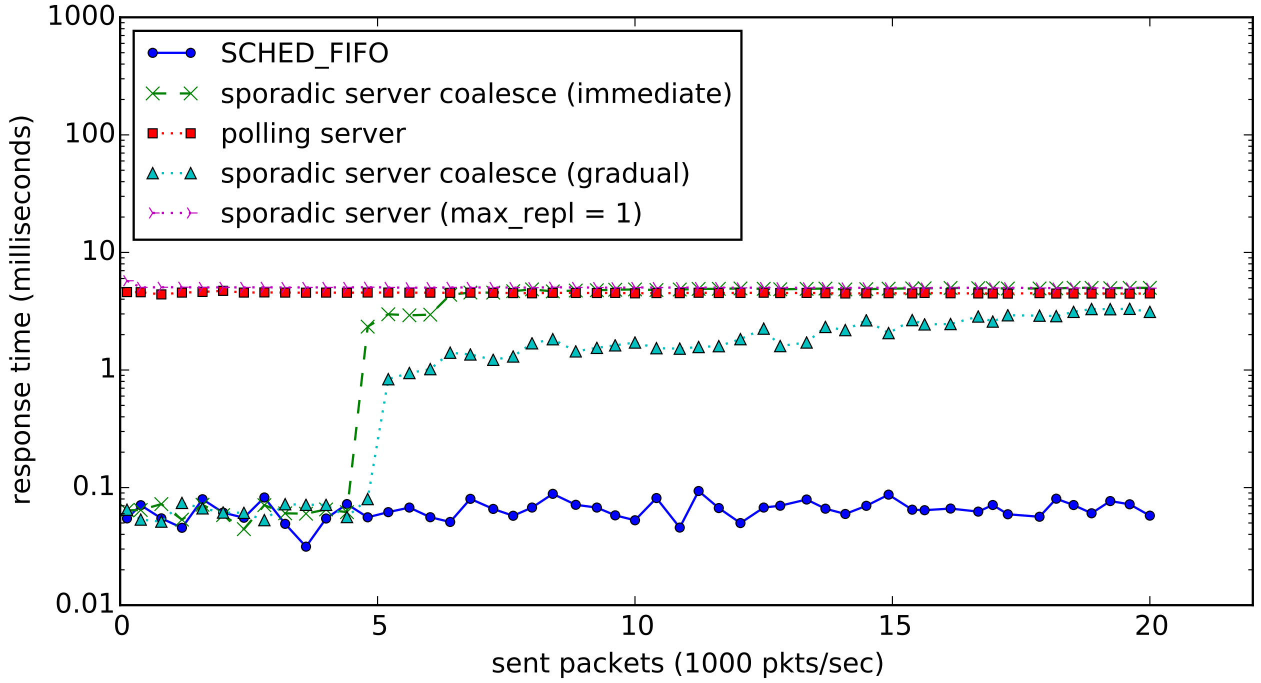


Figure 6.18: Effect on response time when coalescing replenishments under heavy load.

6.7 Related Work

Other works have proposed to improve the original sporadic server algorithm in [106], and at least one other proposed a correction for the error in the original sporadic server definition. However, to the best of our knowledge, this work is the first to point out and provide corrections for the defects in the POSIX sporadic server scheduling policy that are mentioned in this paper.

In [68], Jane Liu gives an example of a defect in the original sporadic server defined in [106], which allows a chunk of time to be replenished too soon. However, she does not discuss the POSIX sporadic server, which has different replenishment rules. In fact, the POSIX sporadic server handles her example correctly. Liu provides an alternate version of the sporadic server algorithm, which is conceptually similar to ours and that of [33] in maintaining the budget in separate chunks, each with their own replenishment times, and using the old replenishment time to compute the next replenishment time for the chunk. Liu's algorithm appears to always avoid premature replenishments, even on the example that we found caused trouble for the POSIX sporadic server. The differences between that algorithm and ours are in details that affect implementation overhead. Liu's algorithm applies a more aggressive and computationally more complex rule for computing replenishment times, based on keeping track of the starting times of busy intervals for each priority. It is less aggressive in attempting to merge chunks and so may require more storage and timer interrupts. It also fails to address the practical matter of budget overruns.

In [30], the importance of handling overruns is addressed. The authors propose a 'payback' mechanism that limits the amount of time a server can accumulate when its budget is exceeded.

In addition, the authors of [30] introduce an optimization to reduce the number of timer events by only arming a replenishment timer when the server has work to perform. We have observed that one can further reduce the number of timer events by not only checking if the server has work to perform, but also checking whether the server has budget. If the server has budget, then the need for a replenishment event is not necessary.

While they provide optimizations for the sporadic server algorithm, they do not address the defects we have mentioned in the current paper.

Davis and Burns [23] evaluated the use of periodic, deferrable, and sporadic servers in fixed priority pre-emptive systems. They provide response time analysis for real-time tasks scheduled under those servers. In a later paper, Davis and Burns [24] define a Hierarchical Stack Resource

Policy (HSRP) for global resource access. That policy allows for server overruns and proposes to use the payback mechanism described in [33]. The approach in our paper also adapts the same overrun and payback mechanism.

6.8 Conclusion

The research in this chapter has proposed modifications to the sporadic server scheduling algorithm to make it an effective solution for scheduling I/O-service processor time. The modifications improve the theoretical formulation allowing it to be practically implemented. Further, additional practical enhancements are given to allow processor allocation for servicing I/O to adapt depending on the server's load.

Prior attempts to formulate a practical specification of sporadic server, including versions in the well-known POSIX specification, have come up short due to unintended defects. One of the primary reasons for these defects appears to be a rather large gap between the theoretical assumptions and the practical hardware limitations. That is, the theoretical formulation did not provide a close enough match to real-world hardware and implementation decisions were made without appropriate regard to the theoretical assumptions. In particular, the research presented in this chapter has shown that the POSIX formulation of the SCHED_SPORADIC scheduling policy suffers from several defects, making it inadequate for its intended purposes. If a critical system is trusted to meet deadlines, based on a schedulability analysis in which a SCHED_SPORADIC server is modeled as periodic server model, the consequences could be serious.

One possible reaction to the defects in the POSIX SCHED_SPORADIC policy is to dismiss it entirely. Some have argued that POSIX should be extended to include other fixed-task-priority budget-enforcing policies [13] that have lower implementation complexity. Others may argue that POSIX should be extended to include deadline-based scheduling policies, which potentially allow deadlines to be met at higher processor utilization levels. Regardless, there is a definite need for a standard scheduling policy that enforces time budgets, and many of the same practical considerations (e.g., overruns) will have to be taken into account.

Properly accounting for preemption costs is necessary for properly limiting processor interference. Charging a sporadic server for preemptions is an effective means to limit the CPU interference. The charging for preemptions can be carried out in several ways. We chose an on-line approach where the server is charged when it preempts another thread. Charging the server only when it actually preempts not only bounds the CPU time for other tasks, but allows the server to use its budget more effectively. That is, rather than accounting for the additional interference by inflating the nominal server budget (over the implemented server budget) in the schedulability analysis, we charge the server at run time for the actual number of preemptions it causes. In this way the server's actual interference is limited to its actual CPU time budget, and we do not need to use an inflated value in the schedulability analysis. Since the preemption charges come out of the server's budget, we still need to consider preemption costs when we estimate the worst-case response time of the server itself. However, if we choose to over-provision the server for worst-case (finely fragmented) arrival patterns it actually gets the time and can use it to improve performance when work arrives in larger chunks.

The ability to use small time slices allows a sporadic server to achieve low average response times under light loads. However, under a load of many small jobs, a sporadic server can fragment its CPU time and waste a large fraction of its budget on preemption charges. A polling server, on the other hand, does not experience this fragmentation effect, but does not perform as well as sporadic server under light load. To combine the strengths of both servers, we described a mechanism to transition a sporadic server into a polling-like mode, thereby allowing sporadic server to serve light loads with good response time and serve heavy loads with throughput similar to a polling server. The data for our experiments show that the hybrid approach performs well on both light and heavy loads.

Clearly, there are devils in the details when it comes to reducing a clever-looking theoretical algorithm to a practical implementation. To produce a final implementation that actually supports schedulability analysis, one must experiment with a real implementation, reflect on any mismatches between the theoretical model and reality, and then make further refinements to the implemented scheduling algorithm until there is a match that preserves the analysis. This sort of interplay between theory and practice pays off in improved performance and timing predictability.

The presented experiences in this chapter suggests a potential improvement to the "NAPI" strategy employed in Linux network device drivers for avoiding unnecessary packet-arrival interrupts. NAPI leaves the interrupt disabled so long as packets are being served, re-enabling it only when the network input buffer is empty. This can be beneficial if the network device is faster than the

CPU, but in the ongoing race between processors and network devices the speed advantage shifts one way and another. For our experimental set-up, the processor was sufficiently fast that it was able to handle the interrupt and the *sirq-net-rx* processing for each packet before the next arrived, but the preemption overhead for doing this was still a problem. By waiting for several packets to arrive, and then processing them in a batch, the polling server and our hybrid server were able to handle the same workload with much less overhead. However, the logical next step is to force a similar waiting interval on the interrupt handler for the network device.

While deadline-based aperiodic servers have not explicitly been considered, it appears that our observations regarding the problem of fitting the handling of preemption overheads to an analyzable theoretical model should also apply to the constant bandwidth server, and that a similar hybrid approach is likely to pay off.

CHAPTER 7

HARD DISK REQUEST SERVICE TIMES

7.1 Introduction

Some form of worst-case service times are used in nearly all real-time schedulability analyses. In the previous chapters, we have focused on CPU worst-case service times since the I/O service being considered, network I/O, had response times primarily dependent on the timely allocation of CPU time. However, CPU time is not the primary contributing factor for timely I/O service in all cases. Therefore, to examine I/O response time, and in particular, the trade-off between latency and throughput, the remaining chapters consider hard disk I/O to illustrate the effects of balancing latency and throughput on meeting timing constraints.

7.2 Overview of a Hard Disk

A hard disk is a device that stores non-volatile, digital data through the use of magnetization of ferromagnetic media [2]. While hard disks have evolved throughout the years, the basic physical design has generally remained unchanged for over a decade. A typical layout of the internal components of a disk drive is shown in Figure 7.1. It is composed of one or more circular, flat disks generally referred to as *platters*. These platters are stacked, with vertical spacing in-between, and rotated about a shared spindle situated through the center of all platters.

Data is stored on platter surfaces, as shown in Figure 7.2. Each recorded surface is read/written by passing a *disk head* over a given data location and magnetizing (write) or detecting the magnetic encoding (read). Generally both surfaces of a given platter are used to store data, however, this is not always the case [2]. While theoretically possible, accessing data on different platters at the same time is generally not supported. The rationale being that it is very difficult to ensure that heads are perfectly aligned on the corresponding tracks [91].

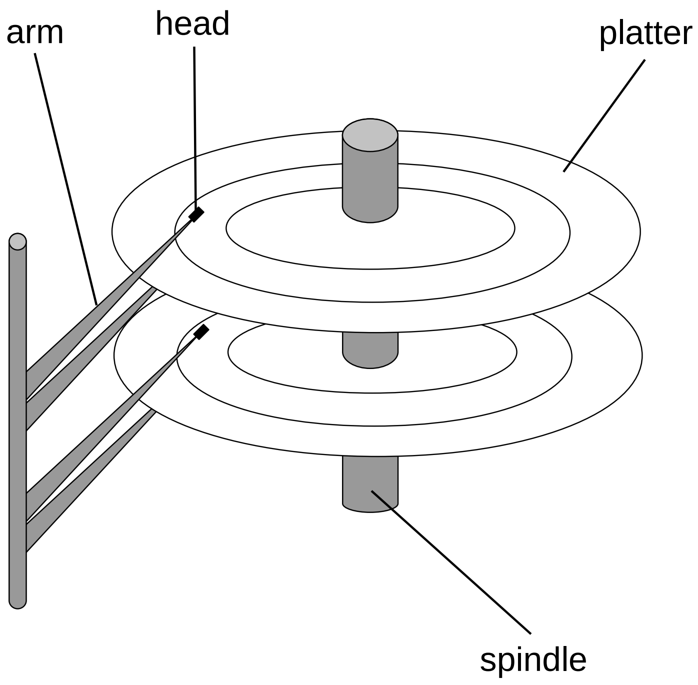


Figure 7.1: Illustration of hard disk internal components.

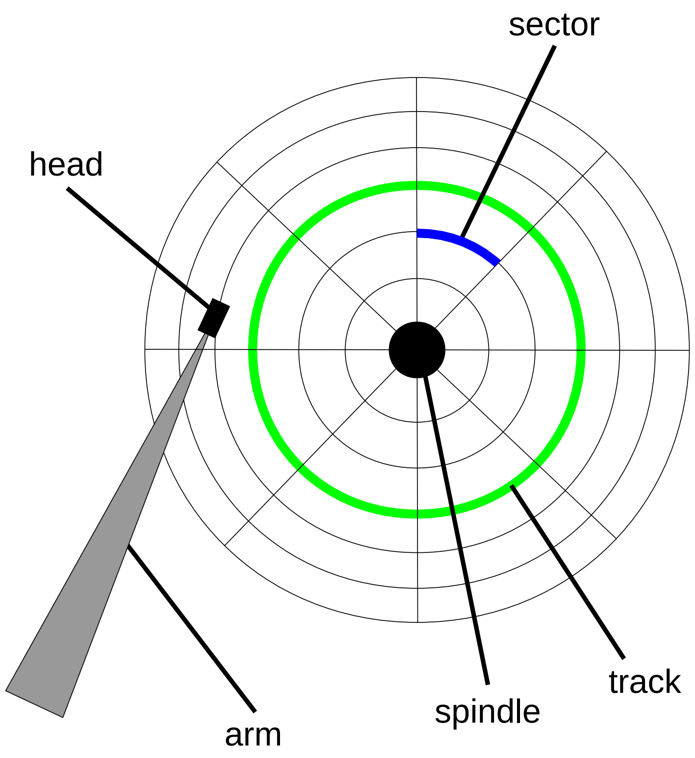


Figure 7.2: One side of a hard disk platter.

The data locations on a hard disk are divided into sectors (sometimes referred to as blocks), where a sector is the granularity for specifying data operations through the disk's interface. These sectors are arranged in multiple concentric circles. Each individual ring of sectors is known as a track. Tracks can be thought of as being similar to the rings seen in a horizontal cross section of a tree. All of tracks with the same radius on every surface are referred to as cylinders.

To address a particular sector the 3-tuple of cylinder, head, and sector (CHS) was originally used. However, modern data hard disks generally only allow specifying locations by logical block address (LBA) (although it is still generally possible to query the hard disk for the CHS location of a LBA). With the logical block addressing, the sectors of a hard disk are logically represented as a one-dimensional array and the LBA is an index into this one-dimensional array.

7.2.1 Operating System Abstraction

The variability of service times for a given request is orders of magnitude larger when compared to most other electronic components of a computer. This variability is primarily due to the timing effect of operations on the state of the hard disk. In particular, the time to service a given request is dependent on the position of the disk head with respect to the data locations composing the I/O request. Therefore, performance of I/O requests is heavily dependent on the service order of requests. For many decades, the I/O scheduler component of an operating system has been responsible for providing request orderings with the aim to achieve high throughput and low average response time while avoiding starved requests.

Typical operating systems have a vague notion of a given disk's hardware characteristics; however, the detailed data layout and capabilities of the disk are largely hidden. Identifying particular data locations on a disk is done through a logical block interface (LBA) that presents data as a one-dimensional array of fixed-sized data units. As such, commonly accepted intuition suggests that the disk's LBA start from the outside perimeter of the disk and progress inwards, but [90] has observed that LBA 0 on some Maxtor disk drives actually starts on track 31. Another example is the popular assumption that issuing requests consecutively increasing/decreasing LBAs will result in the best performance. Again, this is not always true. Some disks support *zero-latency access*, which permits the tail end of a request to be accessed before the beginning. Such a capability enables the disk head to start transferring data as soon as a part of the request is under the disk head, not necessarily starting at the first data location and continuing sequentially to the last. This out-of-order data access scheme reduces the rotational delay when waiting for the beginning of a data request to be positioned under the disk head before beginning a data transfer [119].

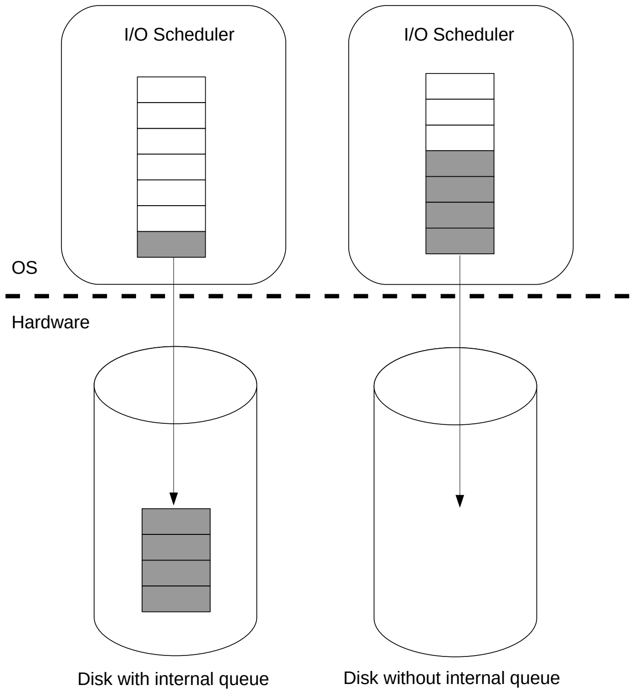


Figure 7.3: Disk-request queuing schemes.

Given that the operating system has limited knowledge of the layout, capabilities, timing characteristics, and the real-time state of any given disks, disk manufacturers provide tailored optimizations such as built-in schedulers to better exploit vendor-specific knowledge of a particular disk. To allow for on-disk scheduling of requests, drives provide an internal queue that allowing storing multiple requests issued from an operating system. The next request to service can then be chosen by the disk's built-in scheduler using specific knowledge for a particular disk drive. Figure 7.3 illustrates the primary difference between disks with an internal queue (and scheduler) and those without. Instead of storing all requests in the OS I/O scheduler framework, disks with a built-in queue allow multiple requests to be issued to the disk without waiting for the completion of a previous request. This permits multiple requests to be pending at the operating system level as well as the hardware level. Both locations allow reordering of requests; however, once requests have been sent to the disk drive, control over their service order shifts from the operating system to the built-in disk scheduler.

Although on-disk schedulers have been shown to provide substantial overall performance increases, they can actually introduce decreased performance for real-time systems. Since the disk can change the ordering of requests, the individual request service times can be difficult to control and accurately predict from the viewpoint of an operating system. Instead of having to consider the worst-case completion time of a single request, a real-time system needs to consider the worst-case completion time of multiple requests which whose service order may be unknown. Therefore, the worst-case service time of requests sent to the disk with an internal queue can be increased to many times that of a disk that does not contain an internal queue. These considerations need to be understood in order to use these disks to meet I/O timing constraints.

7.3 Service Time of a Single Request

Data operations (read/write) are dispatched to a hard disk formatted as requests, *req*. The data location of a *req* is specified by a starting LBA and a *count* of sectors. The starting LBA, *req.start* specifies the first sector and the count, *req.count* specifies the number of sectors starting with the sector *req.start* [115].

The service time of a *req* is largely dependent on the time required to position the disk head over a set of target data locations. The alignment of the disk head over a target data locations generally requires at least two operations: seek and rotation.

The seek operation for a *req* moves the disk head (attached to the disk arm) from its initial position to the track containing *req.start*. A seek operation traversing the largest radial distance (i.e., from the innermost cylinder to the outermost cylinder or vice versa) is referred to as a *full-stroke* seek. The full-stroke seek time refers to the time to move the disk head a full-stroke seek distance.

With the disk head positioned on the track containing the *req.start* address to be accessed (read/written), the continuous rotation of platter results in *req.start* being positioned under the disk head. The rotational delay for serving a request can be divided into positioning and transfer times. The positioning time refers to the time taken to rotate the platter such that the first sector of the request is under the disk head. The maximum positioning delay occurs when the first LBA of the request is just missed when the disk head seeks to the target track. Therefore positioning the disk head over the first LBA is may be delayed by a full rotation, t_{rotate}^{max}. The rotational transfer time, $t_{transfer}$ refers to accessing the contiguously located target sectors. The maximum transfer time, $t^{max}_{transfer}$, refers to the time to access the largest request, req_{max} (contiguous sectors). $t^{max}_{transfer}$ is calculated by dividing req_{max} by the fewest number of sectors on any given track.

Some modern hard drives are able to serve sectors of a request out of contiguous order. This feature is known as *zero-latency* [100]. If the disk head is initially positioned over a data location that is part of the request, data access can proceed immediately, rather than waiting for the first sector of the request. The beginning portion of the operation is served once the portion of the request rotates under the disk head. Therefore, if a request is entirely stored within a single track, a maximum of one rotation is required.

7.4 Expected Service Times

Table 7.1: Maxtor Hard Disk Characteristics.

parameter	**value**
sector size	512 bytes
total sectors	143666192 sectors
average sectors per track	892 sectors
fewest sectors per track	570 sectors
largest sectors per track	1170 sectors
tracks	161111 tracks
cylinders	80568 cylinders
heads	2 heads
platters	1 platter
read cache	disabled
write cache	disabled

The following investigates the expected statistical service times in order to provide some insights into the timing characteristics of a hard disk. While at first, the expected timing characteristics may not seem helpful for reasoning about the maximum service time, some portions of a request's expected service time are directly related to its maximum counterpart. For instance, the mean seek time should approach the expected seek time for a large number of random service time measurements. The expected seek time is approximately 1/3 the full-stroke seek time. Table 7.1 summarizes the hardware characteristics and configuration of our Maxtor drive. Additionally, Table 7.2 lists some timing characteristics. A discussion of these timing values are presented in this section.

Table 7.2: Request service time breakdown for the Maxtor hard drive.

parameter	**origin of value (msr = measurements)**	**notation**	**time (msec)**
full-stroke seek time	linear upper bound of msr	t_{seek}^{max}	8.44
minimum seek time	linear upper bound of msr	t_{seek}^{min}	2.85
expected seek time	derived from msr	$E[t_{seek}]$	4.55
time per rotation	datasheet	t_{rotate}^{max}	6.0
expected rotational positioning	derived from datasheet	$E[t_{rotate}]$	3.0
expected xfer time (255 KiB)	derived from msr	$E[t_{transfer}]$	2.57
max transfer time (255 KiB)	derived from msr	$t_{transfer}^{max}$	5.34

The following discussion assumes that a large number of $255KiB$ requests are sent directly without an underlying file system and randomly with a uniform distribution across all sectors of the disk as performed by our measurement experiments. In order to compare the analytical expected values with that of an actual hard disk, service times were measured on the Maxtor hard disk for 100,000 requests. A histogram of all service times for each request dispatched is shown in Figure 7.4.

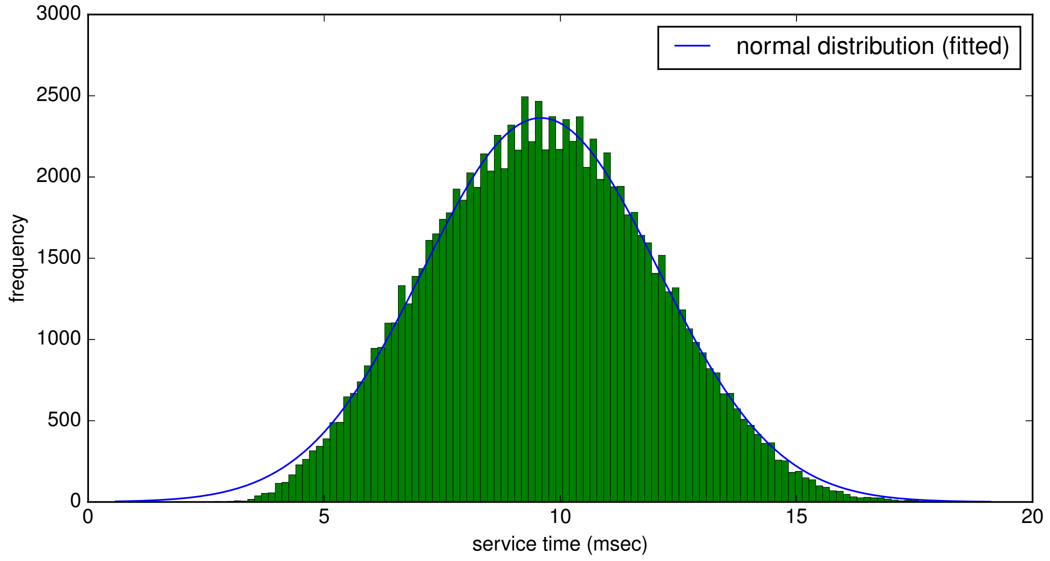


Figure 7.4: 100,000 observed service times of a single 255 KiB read request.

The average service time of a large number of hard disk requests issued randomly is anticipated to be the sum of the expected seek time, rotational positioning, and transfer time. In the case of our Maxtor hard disk, the measured mean for the data in Figure 7.4 is $9.63 msec$, which is fairly close to the sum of the expected seek, rotation, and transfer times is $10.12 msec$. The following discusses the analytical derivation of the expected time calculations used for this comparison.

Expected Rotational Positioning Time. The expected rotational positioning time is 1/2 the time for a full rotation. Let us assume that any location on a track can be the initial head position once positioned on a given track. Therefore, the distance to the first sector of a given request will be modeled as random variable X. Any initial head position is assumed to be equally likely once the disk seeks to the target track. As such, the expected distance to the first sector can be computed by integrating over the track length multiplied by the $1/track.len$. Any distance from 0 to the length of the track is equally likely. The result is one-half the track length, as follows,

$$E[rotate.dist] = \int_{0}^{track.len} x \cdot \frac{1}{track.len}\, \mathrm{d}x = \frac{track.len}{2} \tag{7.4.0.1}$$

Given that the rotational speed is constant (1 revolution per track), it follows then that the expected rotational position time is $t_{rot}^{max}/2$.

Expected Seek Time. The expected seek-distance of a request is approximately 1/3 the full-stroke distance. To compute the expected value of seek distance, the authors of [5, Chapter 37] and [51] provide similar derivations by computing the integral over a continuous function for the seek locations. However, if the seek locations are assumed to be discrete, their analysis is slightly pessimistic. That is, the 1/3 is slightly more than the actual discrete result. The following derives the expected seek distance using the above mentioned authors' derivation, but uses summation properties of finite calculus rather than approximating the summation with a continuous integral.

Let $dist(x, y)$ be the function for the seek *distance* from cylinder x to cylinder y, which can be computed by $|x - y|$. Let $N + 1$ be the total number of cylinders. Further, let X and Y be random variables uniformly distributed over the integers $0, 1, ...N$, representing two cylinders. The expected distance from X to Y can be computed as follows

$$E[dist(X, Y)] = \sum_{x=0}^{N} E[dist(X, Y)|X = x] \cdot P(X = x) \tag{7.4.0.2}$$

where $P(X = x)$ is the probability that X takes the value of x

We assume that the probability of any cylinder x being chosen is uniformly distributed,[1] $P(X = x) = \frac{1}{N+1}$, giving

$$E[dist(X,Y)] = \sum_{x=0}^{N} E[dist(X,Y)|X = x] \cdot \frac{1}{N+1} \tag{7.4.0.3}$$

Factoring out the common $\frac{1}{N+1}$ factor from the summation gives

$$E[dist(X,Y)] = \frac{1}{N+1} \sum_{x=0}^{N} E[dist(X,Y)|X = x] \tag{7.4.0.4}$$

We will now expand $E[dist(X,Y)|X = x]$. By definition of the expected value and the *dist* function, $E[dist(X,Y)|X = x]$ can be written as

$$E[dist(X,Y)|X = x] = \sum_{y=0}^{N} |x-y| \cdot \frac{1}{N+1} \tag{7.4.0.5}$$

Factoring out the common $\frac{1}{N+1}$ factor from the summation gives

$$E[dist(X,Y)|X = x] = \frac{1}{N+1} \sum_{y=0}^{N} |x-y| \tag{7.4.0.6}$$

Expanding the absolute value into two summations

$$E[dist(X,Y)|X = x] = \frac{1}{N+1} \left(\sum_{y=0}^{x} (x-y) + \sum_{y=x}^{N} (y-x) \right) \tag{7.4.0.7}$$

Simplifying the summations using the summation properties from finite calculus [37, Chapter 2] gives

$$E[dist(X,Y)|X = x] = \frac{1}{N+1} \left(x^2 - Nx + \frac{N(N+1)}{2} \right) \tag{7.4.0.8}$$

Substituting this result back into Equation 7.4.0.4 gives

$$E[dist(X,Y)] = \frac{1}{N+1} \cdot \frac{1}{N+1} \left(x^2 - Nx + \frac{N(N+1)}{2} \right) \tag{7.4.0.9}$$

After simplifying gives,

$$E[dist(X,Y)] = \frac{N\,(N+2)}{3\,(N+1)} \tag{7.4.0.10}$$

[1]The assumption that any cylinder is chosen with uniform probability is not exactly accurate. The number of sectors per track (and therefore cylinders) is not the same across all tracks/cylinders due to zoned-bit recording. Therefore, if requests are chosen with uniform probability across all sectors on the disk, then the cylinder corresponding to the chosen request is not precisely uniformly probable across all cylinders.

As anticipated, the expected seek distance is approximately 1/3 the full-stroke seek distance for large number of cylinders. The expected full-stroke seek *time* for 1/3 the full-stroke seek distance is $4.77 msecs$, taken from the seek distance vs seek time measurements, Figure 7.5. A more accurate expected seek time can be computed by using the seek time vs distance data used to plot Figure 7.5. More precisely, let $\Delta_{seek}(x, y)$ be the time to move the disk arm $|X - Y|$ cylinders. The expected seek time calculated as follows:

$$E[t_{seek}] = \sum_{x=0}^{N} E[\Delta_{seek}(X, Y)|X = x] \cdot P(X = x) \tag{7.4.0.11}$$

Using the seek distance vs time measurements, the above equation results in an expected seek time is $4.55 msecs$.

Expected Transfer Time. The expected transfer time must consider that most modern hard disks do not have the same number of sectors per track, which is commonly referred to as zone-bit recording. Therefore, the expected transfer time is:

$$E[t_{transfer}] = \sum_{t=1}^{N} \left(\frac{req.len}{t_i.len} \cdot t_{rotate}^{max} \right) \cdot P(track = t_i) \tag{7.4.0.12}$$

where N is the total number of tracks on the disk and $req.len$ is the data size of the considered request and the probability $P(track = t_i)$ is calculated by dividing the number of sectors on a track t_i by the total number of sectors on the disk. Computing the expected transfer time for the Maxtor hard drive results in $2.57 msecs$.

7.4.1 Service Time of a Group of Requests

The following reasons about an upper bound on the maximum time to serve a given batch of requests assuming requests are served in an optimal or near optimal order.

Many proposed hard disk scheduling algorithms attempt to minimize the total amount of time to service a given set of requests. However, finding an optimal service order to minimize the service time for a set of requests has been shown to be NP-complete by reducing it to the asymmetric traveling salesman problem [3]. [116] has demonstrated that delays due to the processing time to find an optimal solution for I/O request sets greater than 10 result in lower throughput than even FIFO schedulers.

In order to determine offline schedulability, some form of a worst-case I/O service time bound is generally required. While optimal I/O request service orders may improve online performance (e.g., reduced response times), there is often less of an effect for offline schedulability analyses. The reason being that even when the time instant a request is released may be known, the request locations are not known until the time of the request.

The improved real-time performance presented in this research leverages reduced service time of groups of I/O requests relative to the sum of service time if requests in the group were served individually. In the case of a hard disk, the primary reduction in service time of a group is due to a reduction in the total seek time of a group. A hypothesized upper bound on seek time for a set of requests dispatched as a group to a hard disk is simply the sum of worst-case seek times for each individual request. That is, an upper bound on the total seek time for n requests is $n * t_{seek}^{max}$. However, a service order can be found to serve requests in a batch that requires no more than a full-stroke seek distance (not considering the initial disk head position). Considering only seek distance, the problem of finding an optimal solution reduces to the traveling salesman problem on a straight line. The obvious solution is simply a straight line traversal [88]. Serving a group of requests can therefore be achieved by seeking a distance of no more than the number of cylinders on the disk. (i.e., the number of cylinders traversed in a full-stroke seek). Note that the initial location of the disk head is addressed later in this section.

While the maximum *distance* to serve all requests can be accomplished in a single full-stroke sweep, the maximum *time* to visit all requests will be more than the full-stroke seek time. A non-linear relationship exists between seek distance and seek time. In order to generate a seek curve, the DIXTRAC [31] software was used. To generate the seek time vs distance plot, DIXTRAC issues several I/O seek requests from random cylinders and measures the time to seek the specified distance in both directions (towards the center of the platter and towards the outside of the platter) [99]. The seek distance vs time curve for the Maxtor disk is shown in Figure 7.5.

Consider the following I/O scheduling algorithm. Evenly divide the cylinders of the disk into the innermost half and the outermost half. Let the initial position of the disk's head be at $head_{init}.cyl$. Any distances are assumed to be in terms of seek distances measured in cylinders. Given a set of requests, the first request to be served is chosen based on one of the following two cases:

(1) One or more requests exists on the same cylinder half as $head_{init}.cyl$:

The first request served is the request nearest the edge, inner or outer, on the same half as $head_{init}.cyl$.

(2) No request exists on the same cylinder half as $head_{init}.cyl$:

The first request served is nearest to $head_{init}.cyl$

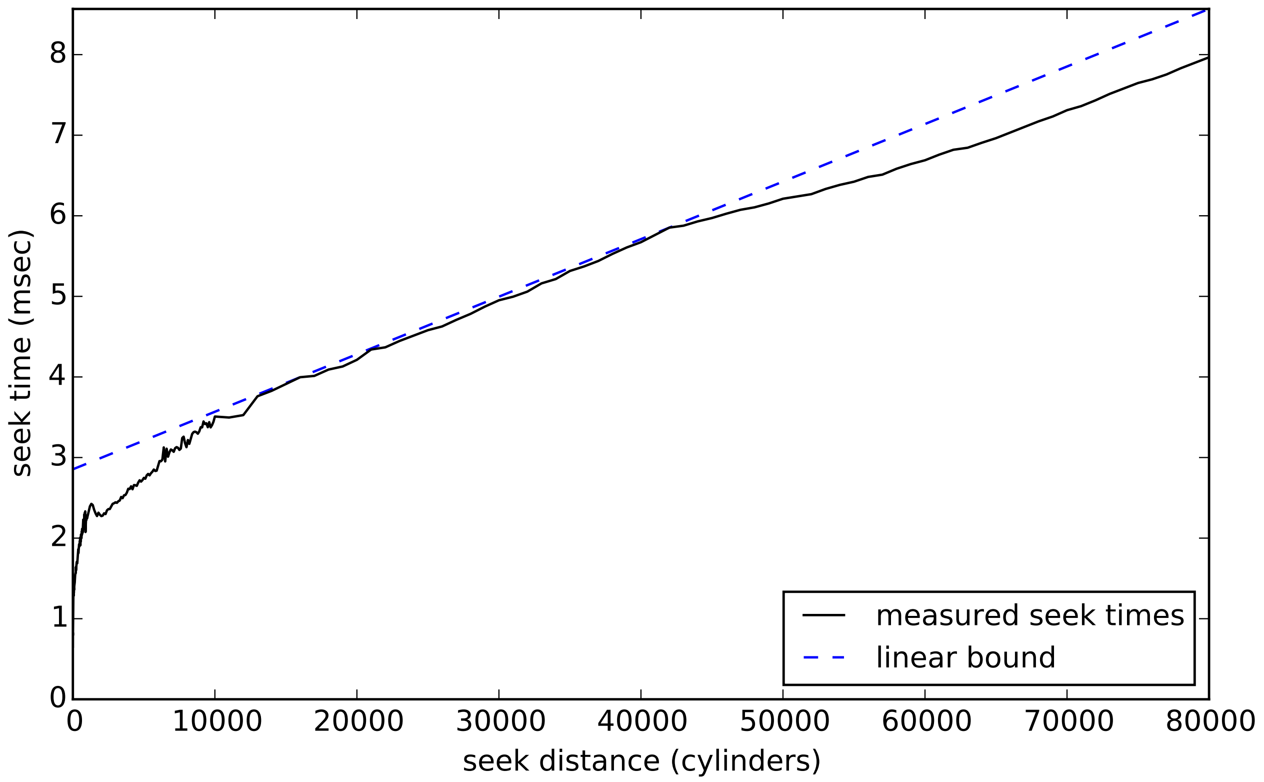


Figure 7.5: Seek distance vs seek time curve for Maxtor hard drive.

The remaining requests are served in order of cylinder number (either increasing or decreasing) in order to serve all requests by moving the disk head in only one direction.

Assuming the above scheduling algorithm for service order of requests in a batch, the following derives an upper bound on the total seek time. Compute a linear upper bound on seek time for all possible seek distances as shown in Figure 7.5. The equation of this linear bound is,

$$\Delta_{seek}(head, req) = m \cdot (|head.cyl - req.cyl|) + t_{seek}^{min} \tag{7.4.1.1}$$

($|head.cyl - req.cyl|$ is simply the seek distance to for serving req)

Let $req_i.cyl$ and t_{seek_i} be the cylinder of the i^{th} request and the time to seek to the i^{th} request from the $i-1^{th}$ request respectively ($req_0.cyl$ is the initial head position). The total seek time of a given batch is the sum of the seek times to each request in the batch.

$$t_{seek_1} + t_{seek_2} + \ldots + t_{seek_n} \tag{7.4.1.2}$$

where

$$\begin{aligned}
t_{seek_1} &= m(|req_1.cyl - head_{init}.cyl|) + t_{seek}^{min} \\
t_{seek_2} &= m(|req_1.cyl - req_2.cyl|) + t_{seek}^{min} \\
t_{seek_3} &= m(|req_2.cyl - req_3.cyl|) + t_{seek}^{min} \\
&\vdots \\
t_{seek_n} &= m(|req_n.cyl - req_{n-1}.cyl|) + t_{seek}^{min}
\end{aligned}$$

Based on the above scheduling algorithm, the seek distance from req_1 to req_n is no more than a full-stroke seek. Therefore, $|req_n.cyl - req_1.cyl| \leq disk.cyl_{total}$. Further, an additional seek of at most $\frac{head.cyl}{2}$ is required in the first case described above when choosing the first request of the batch to serve. So, the above sequence can be reduced to the following:

$$m(1.5 \cdot disk.cyl_{total}) + n \cdot t_{seek}^{min} \tag{7.4.1.3}$$

m is the slope of the linear upper bound on the seek curve, so $m = \frac{t_{seek}^{max} - t_{seek}^{min}}{disk.cyl_{total}}$. Substituting for m in the above equation gives

$$\frac{t_{seek}^{max} - t_{seek}^{min}}{disk.cyl_{total}} \cdot (1.5 \cdot disk.cyl_{total}) + n \cdot t_{seek}^{min} \tag{7.4.1.4}$$

canceling common $disk.cyl_{total}$ factors leaves

$$1.5 \cdot (t_{seek}^{max} - t_{seek}^{min}) + n \cdot t_{seek}^{min} \tag{7.4.1.5}$$

SCAN-EDF [92] derives an upper bound of on the seek time for a batch to be $2 \cdot (t_{seek}^{max} - t_{seek}^{min}) + (n+1) \cdot t_{seek}^{min}$. Our derived upper bound established in the above Equation 7.4.1.5 is a slight refinement. The improvement is achievable by requiring only 1/2 seek rather than a full-stroke seek for the head movement to serve the first request. Further, one minimum seek term is removed given that the initial seek can be used to move directly to the cylinder of the first served request.

Unfortunately, most disks do not appear to adhere to our hypothesized scheduling algorithm. The disks used for our experimental evaluation appear to generally use a SCAN and SSTF hybrid, where the first request is chosen to be the request nearest the disk head. Subsequent requests are served in the same SCAN order. That is, all requests are served in two disk head sweeps: first those reachable in the same direction as the first request followed by the remaining requests reachable by moving the disk head in the opposite direction. While a single request would still be 1 full-stroke seek, 2 or more requests for such a scheduling algorithm results in a seek distance of 2 full-strokes and a maximum seek time as follows:

$$\begin{cases} t_{seek}^{max}, & \text{if batch size} = 1 \\ 2 \cdot (t_{seek}^{max} - t_{seek}^{min}) + n \cdot t_{seek}^{min}, & \text{if batch size} \geq 2 \end{cases} \tag{7.4.1.6}$$

The above equation more accurately models the hard disks used in our experiments and therefore is used for the corresponding analysis. Certainly further refinements are possible. For instance, the linear upper bound on seek time is conservative and therefore, a tighter bound is possible by using a more precise approximation as described by Oyang [84]. However, such refinements are likely to overly complicate our discussion with little gain and therefore are not included.

In addition to seek time, as mentioned previously, serving a given request entails some amount of rotational latency. For our service time calculation, the rotational latency is divided into the maximum of a full rotation for positioning (t_{rotate}^{max}) and transfer ($t_{transfer}$) time. Adding these terms comprising rotational latency to Equation 7.4.1.6 gives

$$\begin{cases} t_{seek}^{max} + t_{rotate}^{max} + t_{transfer}, & \text{if batch size} = 1 \\ 2 \cdot (t_{seek}^{max} - t_{seek}^{min}) + n \cdot (t_{rotate}^{max} + t_{transfer} + t_{seek}^{min}), & \text{if batch size} \geq 2 \end{cases} \tag{7.4.1.7}$$

7.4.2 Measured vs. Theoretical Maximum Service Times

The previous analyses for analytically reasoning about the maximum service time are at best loose approximations of the true worst-case. For instance, consider the previous experiment of Figure 7.4, where the service times of 100,000 randomly issued requests were each individually measured. The longest measured service time is $29.9msecs$, which is in contrast to an analytical worst-case approximation of $t_{seek}^{max} + t_{rotate}^{max} + t_{transfer}^{max} = 19.78msec$. However, the $29.9msec$ request only occurred once in the 100,000 measured requests. The second largest value was 20.9$msecs$. Therefore, measuring service times seems to be a necessary approach compared to solely using the data sheet values. In fact, some hard disk manufacturers state that they use empirical measurements for timing characteristics. As an example, the Maxtor drive's datasheet [73] states that the full-stroke seek time "... is measured by averaging the execution time of a minimum of 1000 operations ...".

One may consider using a lower maximum service time for the response time analyses, if there is sufficient evidence indicating that the longer service times are truly infrequent occurrences. Others [86] have taken a similar approach to deal with "outliers," by reasoning that they can be accommodated by adding small overhead terms. A potential issue with this approach is the difficulty of determining the actual underlying frequency model of these outliers. It is not immediately obvious what triggers these outliers and so it is conceivable that an application workload could continuously trigger these outliers, potentially resulting in a large number of missed deadlines.

A back-of-the-envelope statistical reasoning for choosing a given number of service times measurements may provide some confidence in our choice of 100,000 measurements. Initially we chose 100,000 since the performing this number of measurements took a reasonable amount of time to perform (i.e., 1 day). Intuitively, it would appear that the greater the number of observations the better, however, given errors in the experiment and other factors, a limited number of observations is practically possible. The following describes a potentially better approach for fine-tuning the number of observations. From our past experience most worst-case values are approximately 4 standard deviations (4σ) from the mean. Therefore, we would like to be somewhat confident (say 99%) that we will observe a service time greater than one 4σ from the mean. Let q be the probability of observing a service time less than or equal to 4σ greater than the mean on a single request, which we will call t_{max}. Therefore, given k observed service times, the probability that all k are less than or equal to t_{max} is q^k. Subtracting q^k from 1 gives us the probability of observing at least one value greater than t_{max}. Let us set as our goal the probability of 99% to observing at least one value greater than or equal to t_{max}, giving

$$1 - q^k = 0.99 \tag{7.4.2.1}$$

Solving for q gives

$$k = \frac{log(0.01)}{log(q)} \tag{7.4.2.2}$$

If we assume that the service times are normally distributed, the cumulative probability at 4σ is 99.993666%. Therefore, evaluating the above equation gives $k = 72,704$. Therefore, if we make 72,704 measurements we can be 99% confident that we will observe a service above the service time at 4σ from the mean.

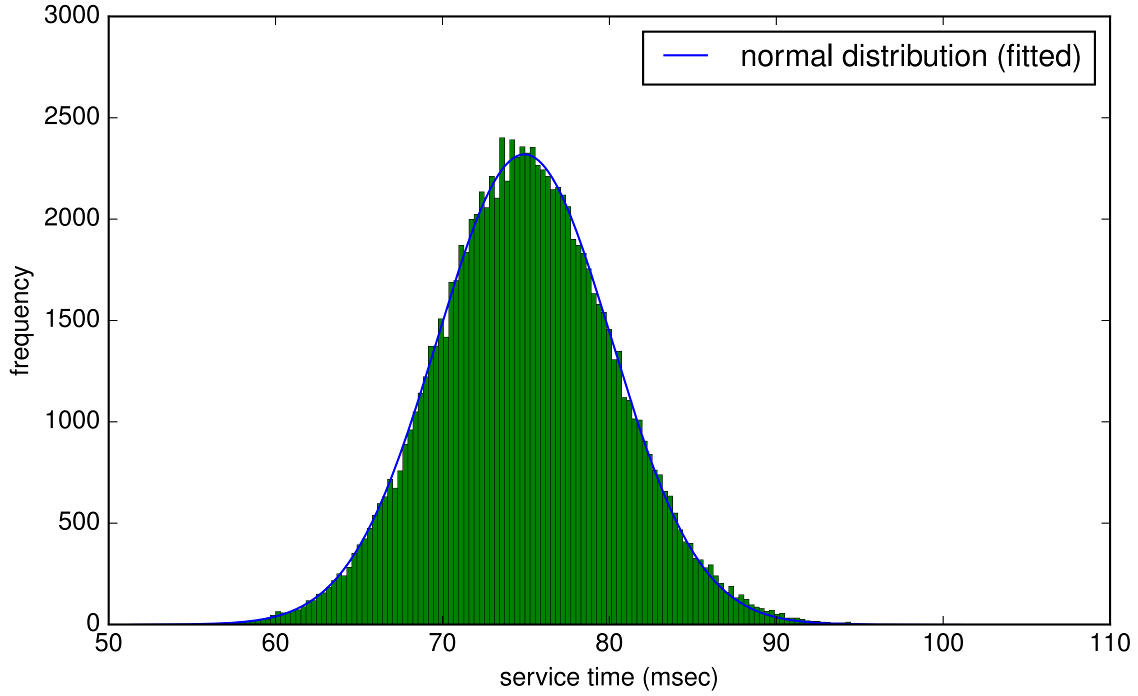


Figure 7.6: Distribution of measured service times for batches of size 10. Each request is a 255KiB read requests.

We also investigated other statistical approaches. In particular, we attempted to gain more confidence in the low probability of these seemingly anomalous, larger service time values by applying the statistical-based worst-case estimation technique described in [40]. Their proposed method uses extreme value theory to predict the probability of exceeding a given execution time on a processor. The application of their method on the service time of a given batch size did not immediately result in reasonable evidence that long service times could be treated as low probability events. The result of the estimated worst-case service time using the statistical technique of [40] is shown in Figure 7.7. Their approach allows one to calculate a service time estimate for a given probability. That is, the estimated service time has the given probability of being exceeded in the future. Unfortunately, the initial results from applying their technique appear to indicate that their technique is not directly applicable our measurements. Our hope was that the predicted execution time would always be greater than the measured execution time for a given probability. However, as seen from the figure, the measured execution time is greater than the predicted. One explanation that the technique does not seem to work on the hard disk is that the measurement distribution does not have the heavier tail of the Gumbel distribution as shown in Figure 7.6.

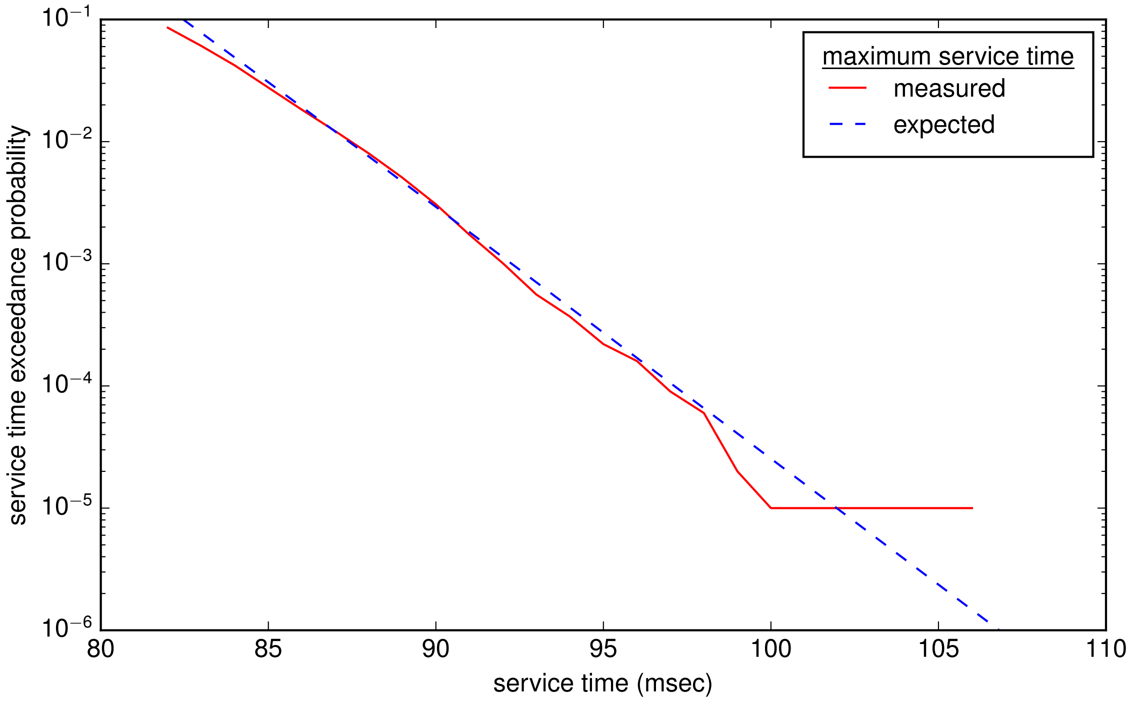


Figure 7.7: Results using the extreme value technique of [40] for service time measurements of batches with 10 requests.

Additional empirical measurements on our Maxtor hard disk [73] were performed to compare them with our hypothesized upper bound on request service times for batches of requests. The results are illustrated in Figure 7.8. The data was generated by simultaneously dispatching a batch of requests (the number of requests in a batch corresponding to the axis labeled load in the figure) of uniformly distributed random read requests of 255KiB (each request was a contiguous with respect to LBAs). The time to service each batch was recorded. The requests were submitted from a user-space application and each data point represents a total of 100,000 individual batch service time measurements. The data labeled with "w/o batching" corresponds to dispatching the batch of requests from the OS one at a time in FIFO order. The "w/ batching" experiments issue all requests at one time to the disk and in this case, the hard disk may (and likely will) serve requests in a different order than FIFO. The longest service time and the service time greater than 99.9% of all measured request service times is given shown in Figure 7.8 The data points corresponding to 100% and 99.9% maximums are indicated in the legend of Figure 7.8 with a trailing (1.0) and (0.999) respectively.

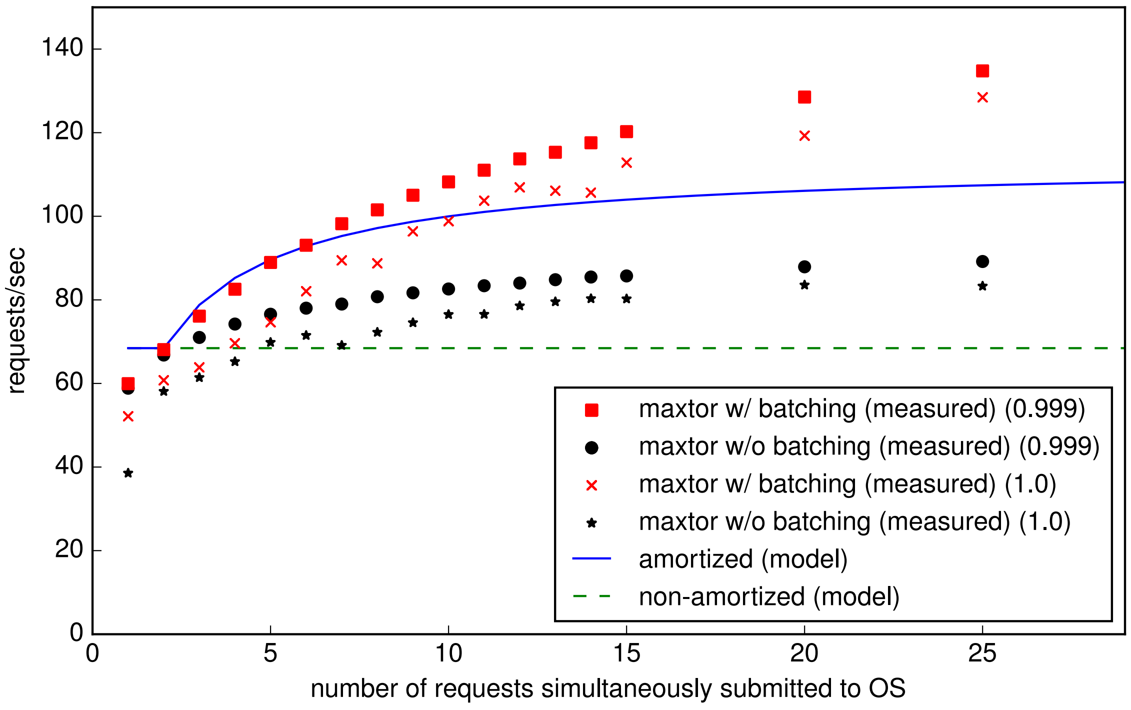


Figure 7.8: Dispatching batches of requests vs one-at-a-time increases throughput.

Figure 7.9 provides another illustration of the measured maximum service time of 100% and 99.9% alongside the distribution of service times. The increase in service time from 99% to 100% is fairly large

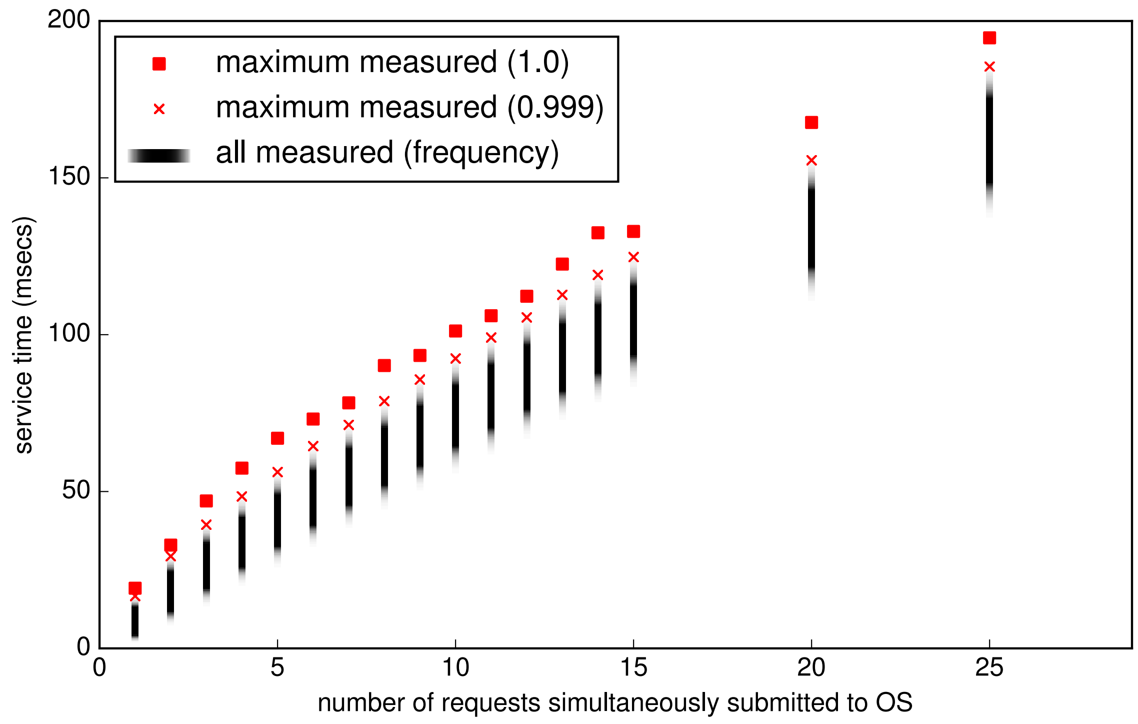


Figure 7.9: Maximum over 100% and 99.9% of all measured service times (w/ batching).

The measured data loosely matches our hypothesized maximum service time model. In general, the measured service times of smaller-sized batches are longer than our model predicts and therefore our model underestimates the actual worst-case. On the other hand, the measured service times of larger batches tend be smaller than our model predicts, meaning that our model may be too conservative. For analysis that relies a worst-case model, a reasonable approach may be to take the worse of the analytical and measured. A conjecture for the differences between our predicted model and the actual observed measurements is that (1) for smaller batches, our model does not fully capture all characteristics of the actual hard disk. Therefore, on an actual hard disk, we were able to observe such characteristics that extend service times, but are not included in our model. (2) Larger batch sizes significantly decrease the probability of observing even a near worst-case. Intuitively, the probability of all requests in a batch resulting in all of them requiring a near worst-case service time is likely to be small. Of course, this is based on the assumption that the request service times are solely dependent on the address of the requests and not other factors that may not be directly related to the request locations.

7.5 Conclusion

Approximate worst-case service times of a commodity hard disk drive can be inferred using a combination of measurement and analytical techniques. Generally speaking, real-time analyses depend on some form of worst-case service times for serving requests. However, the complexity and variability of commodity devices (e.g., different models) makes it difficult to analytically derive exact worst-case service times. Therefore, to provide a reasonable worst-case service we performed experimented with an actual hard disk and measured service times. In order to understand the measured service times, analytical reasoning from high level operations of the disk was presented.

Our general approach is to use the observed measurements rather than the analytically derived worst-case service times for schedulability analyses. While the observed worst-cases may not be the actual worst-case, the observations seem to provide a reasonable approximation of the actual hard disk worst-case service times (both in terms of actual and probabilistic worst-case service times), more so than the worst-cases predicted by our analytical model. One can certainly argue that a more precise analytical model can be derived. However, given the complexity and variability in modern commodity hard disks the practicality of such an approach is questionable. That is, finding a general technique to extract fine-grained characteristics from any given hard disk seems to be unlikely given the diversity and complexity of modern hard disks.

CHAPTER 8

LEVERAGING BUILT-IN SCHEDULERS FOR SINGLE RT TASK TIMING GUARANTEES

8.1 Introduction

This chapter discusses the effect of scheduling peripheral device requests on I/O service in a real-time system. A GPOS generally provides I/O service using a processing unit (i.e., CPU) and a peripheral device. While the previous chapters considered the effect of I/O services on processor time allocation, the following chapters consider the effect of peripheral device scheduling on I/O service performance. To study the effect of peripheral device scheduling, this chapter uses the hard disk drive as an example peripheral device. In contrast to the network service, which require significant amounts of processor time (e.g., network services), I/O services using the hard disk are relatively slow in comparison to the CPU (e.g., hard disk drives) and generate little processor demand. Since small amounts of processor time are required, generally the corresponding thread of I/O service can be simply scheduled at the highest priority of the system with negligible effects on schedulability. However, even with optimal allocation of processor time, unacceptably poor I/O performance may result due to inefficient scheduling of I/O requests to peripheral devices.

Properly scheduling requests to achieve bounded I/O response time with peripheral devices such as a hard disk is often difficult due to characteristics such as relatively lengthy and non-preemptive operations. For instance, a hard disk I/O request with a short deadline may be missed due to another recently started, long, non-real-time I/O request. Further, solely scheduling I/O requests based on deadlines (e.g., EDF) without considering the long-term impacts on throughput often results in missed deadlines due to the lack of throughput. That is, while "near-term" deadlines may be met, without sufficient throughput deadlines will eventually be missed.

The majority of the results in this chapter are part of my Master's thesis and are published in [107]. The research in the subsequent Chapter 9 builds upon the ideas from this work and so it is included to provide background information. In particular, this chapter discusses a different "class" of device and uses the hard disk to emphasize these differences. Further, a scheduling technique called *draining* is introduced which provides a means to guarantee a single, periodic real-time device request in the presence of an unlimited number of non-real-time requests. The limit of a single real-time request is relaxed as part of this dissertation research in the subsequent Chapter 9.

The outline of this chapter is as follows. First, the difficulties with meeting I/O service timing constraints with hard disk drives on a GPOS are given in Section 8.2. A novel scheduling technique termed draining is detailed in Section 8.3 to provide a means to bound response times for a real-time task's I/O service while maintaining high throughput for non-real-time applications. Section 8.4 provides data from a system implementing the draining approach, which demonstrates its effectiveness through comparisons with other techniques. Lastly, Section 8.5 concludes the chapter.

8.2 Obstacles to Meeting Timing Constraints

Certain characteristics of disk drives make it difficult to predict I/O response times accurately. One such characteristic is the variability of service times caused by the state of the disk due to a prior request. With disks that allow one outstanding request at a time, a new request sent to the disk from the OS must wait until the completion of the previous request. Only then can a new request be issued from the OS to the disk. Next, based on the location of the previous request, the disk must reposition its read/write head to a new location. Given these factors, the variability of timings can easily be on the order of tens of milliseconds.

Many disks now also contain an extra state parameter in the form of an internal queue, which is available on most SCSI and SATA disks. To send multiple requests to the hard drive, a *command queuing* protocol is used with three common policy variants: simple, ordered, and head-of-the-queue [115]. Policies are specified with a tag as each request is sent to the disk. The "simple" tag indicates that the request may be reordered with other requests also marked as "simple". The "ordered" tag specifies that all older requests must be completed before the "ordered" request begins its operation. The "ordered" requests will then be served followed by any remaining "ordered" or "simple" tagged commands. Finally, a request tagged with "head-of-the-queue" specifies that it should be the next command to be served after the current command (if it exists).

With an internal queue, the variability in request service time is significantly larger. Once a request is released to the disk for service, the time-till-completion will depend on the service order of the currently queued requests, which is established by both the disk's internal scheduler and the

Table 8.1: Hardware and software experimental specifications.

Hardware/software	**Configurations**
Processor	Pentium D 830, 3GHz, 2x16KB L1 cache, 2x1MB L2 cache
Memory	2GB dual-channel DDR2 533
Hard disk controller	Adaptec 4805SAS
Hard disks	Maxtor ATLAS 10K V, 73GB, 10,000 RPM, 8MB on-disk cache, SAS (3Gbps) [73]
	Fujitsu MAX3036RC, 36.7GB, 15,000 RPM, 16MB on-disk cache, SAS (3Gbps) [64]
	IBM 40K1044, 146.8GB, 15,000 RPM, 8MB on-disk cache, SAS (3Gbps) [42]
Operating system	Linux 2.6.21-RT PREEMPT

given insertion policy. Therefore, in contrast to issuing requests one-at-a-time, where a request may be served in tens of milliseconds, the maximum time to serve a given request when issued to the disk can be increased to several seconds.

8.2.1 Observed vs. Theoretical Bounds

To demonstrate and quantify problems of real-time disk I/Os resulting from drives with internal queues/schedulers, we conducted a number of simple experiments. These tests were run on the RT Preempt version of Linux [65], which is standard Linux patched to provide better support for real-time applications. The hardware and software details used for the following experiments are summarized in Table 8.1.

To estimate service times for a particular request by a real-time application, one could make simple extrapolations based on the data sheets for a particular drive. Considering disk reads, a naive worst-case bound could be the sum of the maximum seek time, rotational latency, and data access time. According to the Fujitsu drive's data sheet [64], the maximum seek time and rotational latency are 9 and 4 milliseconds respectively. Assuming that the disk head can read as fast as the data rotates underneath the head, the data transfer time would then be the time spent rotating the disk while reading the data, which is a function of the request size. For our experiments, we chose a 256KB request size. Since modern drives store more data on outer tracks than inner tracks, the chosen access granularity (256KB) typically ranges from half of a track to slightly more than one track on various disks. Thus, a request could potentially take another rotation to access the data with an extra 4 msec, resulting in a worst-case bound of 17 msec. Clearly, this crude estimation overlooks factors such as settling time, thermal recalibration, read errors, bad sectors, etc. However, the aim is to develop a back-of-the-envelope measurement on the range of expected service times.

To validate the estimated service times empirically, we created a task that issues 256-KB requests to random disk locations. Each request's completion time was plotted as shown in Figure 8.1. The first observation is that almost all requests were completed within the predicted 17 msec time frame. However, a few requests exceeded the expected maximum completion time, the latest being 19 msec. These outliers could be attributable to any of the above mentioned causes that were not included in our coarse estimation method. With the observed worst-case completion time of 19 msec, using disks for soft-real-time applications seems quite plausible.

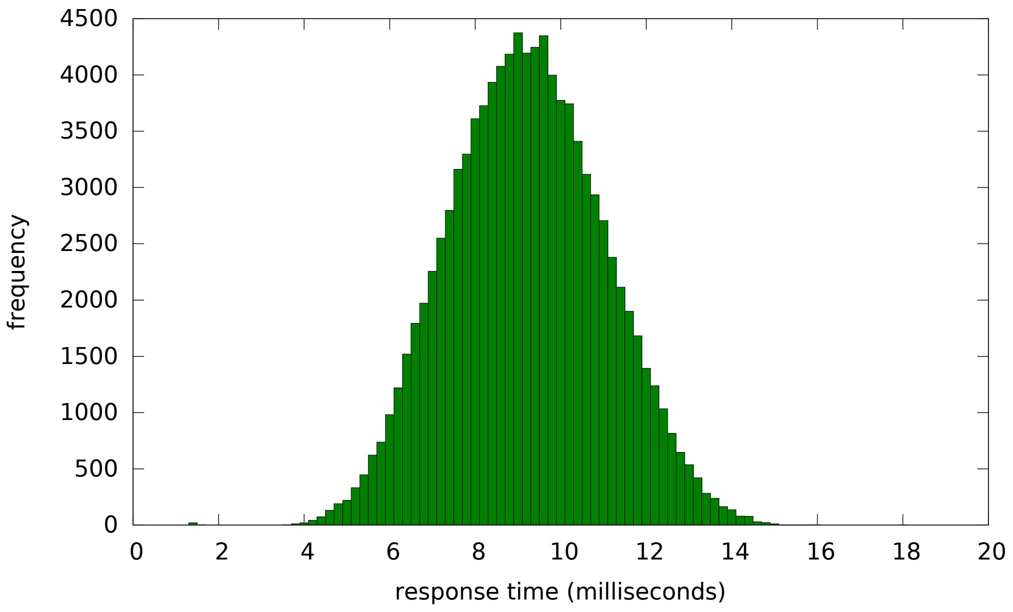


Figure 8.1: Observed disk completion times using Fujitsu HDD with no interference.

8.2.2 Handling Background Requests

The above bound does not consider the common environment with mixed workloads, where real-time requests and best-effort requests coexist. This mixture increases the worst-case completion time for a single request. On a disk that can accept one request at a time, this worst-case completion time for a request could be estimated at twice the maximum completion time for one request, that is: one time period for a request being served, which cannot be preempted and another time period to serve the request. However, disks with internal queues paint a very different picture.

Internal queues allow multiple requests to be sent to the disk from the OS without having to wait for previous requests to be completed. Given several requests, the disk's scheduler is then able to create a service order to maximize the efficiency of the disk to serve requests. This capability, however, poses a problem regarding requests with timing constraints, since meeting timing constraints can be at odds with servicing the given requests efficiently.

Consider a scenario with both real-time and best-effort requests. For real-time requests, the OS I/O scheduler should support real-time capabilities and therefore could send real-time disk requests before any best-effort requests. Without real-time capabilities, an arbitrary real-time request may be delayed by an accumulation of best-effort in the OS queue (e.g., if the OS scheduling policy is FIFO with no awareness of real-time priorities). While Linux does have an interface to set the I/O priorities, only the Complete Fairness Queueing[1] (CFQ) [7, 8] scheduler applies any meaning to the value of these priorities. Modifying the CFQ scheduler is one option we considered; however, due to its complex code base, we felt that it would be very difficult to include. CFQ is not designed for meeting deterministic completion times and has substantial amounts of code that are intended to provide good average-case and fairness performance, but could potentially interfere with our real-time scheduling technique. That is, it would likely be very difficult to have confidence that the final merged result was a faithful representation of our design.

8.2.3 Prioritized Real-Time I/Os are not Enough

To investigate the latencies associated with using disk drives, we implemented a basic real-time I/O (RTIO) scheduler in Linux. This scheduler honors the priorities set by the individual applications. Further, request merging was not permitted, in order to prevent non-real-time requests

[1]CFQ is sometimes called Completely Fair Queueing.

from extending the time to complete a given request. The requests within individual priority levels are issued in a first-come-first-served fashion. All reordering and merging are handled by the disk's scheduler which frees the OS scheduler from the burden of ordering requests and thereby reduces the load on the system's processor.

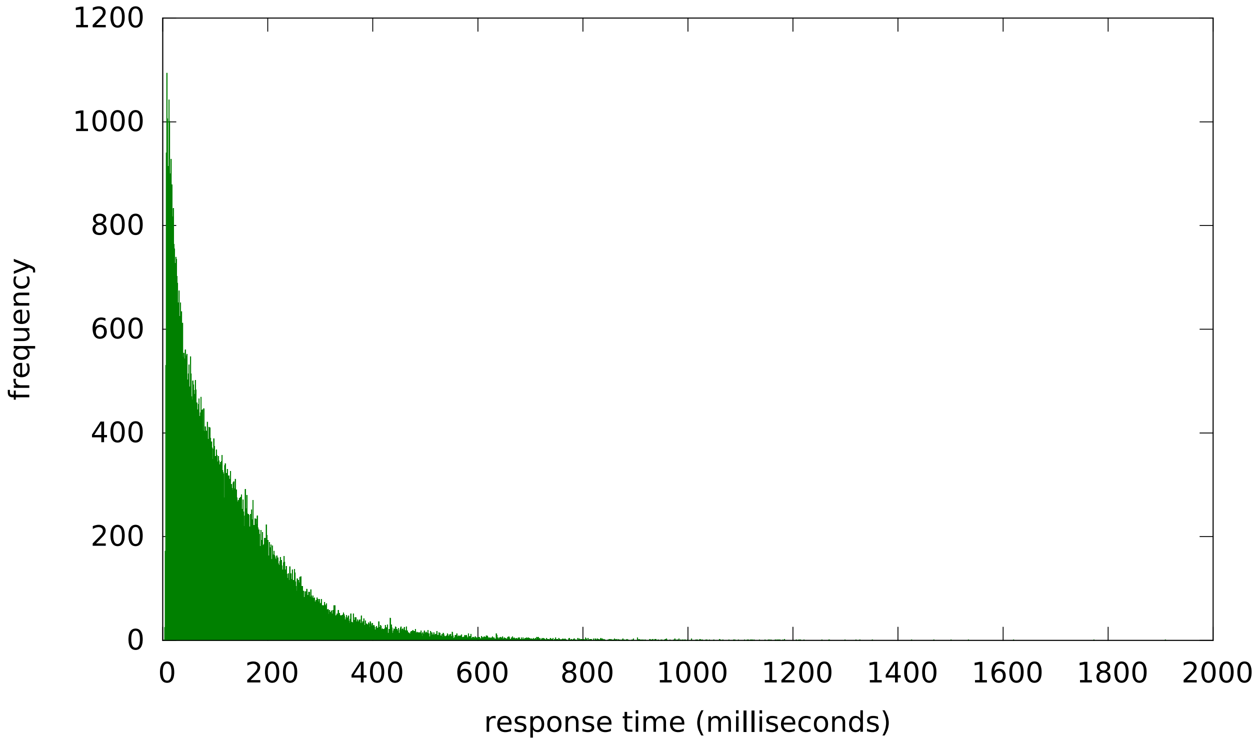


Figure 8.2: Completion times for real-time requests in the presence of background activity using the RTIO scheduler.

Using our RTIO scheduler, we designed an experiment to measure real-time request latencies where two processes generated 256-KB read requests to random locations on disk. One is a real-time task that repetitively issues a read and measures the amount of time for it to complete. The second task is a best-effort multi-threaded process that generates up to 450 read requests continuously. The idea is to generate significant interference for the real-time task in order to make it more likely to measure the worst-case completion time for a real-time request. The completion times of the real-time requests using a Fujitsu drive are graphed in Figure 8.2. As illustrated in the figure, the reordering of requests by the disk can cause unexpectedly long response times with the largest observed latency being around 1.9 seconds. The throughput, however, was respectable at

39.9 MB/sec .[2] With such a workload (i.e., non-real-time and best-effort), there is essentially two conflicting goals; high throughput and low worst-case response time. Although ideally one would like high throughput and low worst-case response time, optimizing both is often not possible. For instance, when making offline timing guarantees the lack of information about online disk request locations limits the ability to make optimal decisions. In the previous experiment, the on-disk internal scheduler seems to favor high throughput.

Using disks with a built-in scheduler can significantly increase the variance of I/O completion times and reduce the ability to predict a given response time as opposed to disks without a built-in scheduler. However, the internal queue does provide some benefits. Not only does it provide good throughput, it also allows the disk to remain busy while waiting for requests to arrive from the device driver queue. Particularly, in the case of real-time systems, the code for the hard disk data path might have a lower priority than other tasks on the system, causing delays in sending requests from the device driver to the disk, to keep the disk busy. Without an internal queue, a disk will become idle, impacting both throughput and response times of disk requests. The severity depends on the blocking time of the data path. Even with an internal queue, the problem of reducing and guaranteeing disk response time remains. Without addressing these issues, it is unlikely anyone would choose to use a disk in the critical path of real-time applications.

8.3 Bounding Completion Times

This section describes the various techniques explored to bound the completion times of real-time requests issued to a hard disk. These include using the built-in starvation prevention algorithm on the disk, limiting the maximum number of outstanding requests issued to the disk, and preventing requests being sent from the OS to the disk when completion time guarantees are in jeopardy of being violated.

[2]The maximum observed throughput of the disk on other experiments in which we fully loaded the disk with sequential read requests ranged from 73.0 to 94.7 MB/sec, depending on the cylinder. Clearly, that level of throughput is not possible for random read requests. A rough upper bound on random-access throughput can be estimated by taking the request size and dividing it by average transfer time, rotational delay, and seek time for 20 requests. For our experiment, this is 256KB/(3 msec (to transfer 256KB) + (4 msec (per rotation)/2) + 11 msec (worst-case seek time)/20), giving a throughput of 46.1 MB/sec. This is not far from the 40 MB/sec achieved in our experiments.

8.3.1 Using the Disk's Built-in Starvation Prevention Schemes

Figure 8.2 illustrates that certain requests can take a long time to complete. There is, however, a maximum observed completion time of around two seconds. This maximum suggests that the disk is aware of starved requests, and it forces these requests to be served even though they are not the most efficient ones to be served next. To test this hypothesis, we created a test scenario that would starve one request for a potentially unbounded length of time. This was achieved by issuing one request with a disk address that is significantly far away from other issued requests. Therefore, servicing the single distant request would cause performance degradation. Better performance would result if the distant request were never served. The measured length of time it takes to serve the distant request is likely to be close to the maximum time a request could be queued on a disk without being served. The intuition is that the larger the number of such sets of requests are issued and timing measured the more likely it is to measure a time near to or at the actual worst-case.

To be explicit, best-effort requests were randomly issued to only the lower 20 percent of the disk's LBAs. At the same time, one real-time request would be issued to the disk's upper 20 percent address space. A maximum of twenty best-effort requests and one real-time request were permitted to be queued on the disk at one time. The idea was that the disk would prefer to serve the best-effort requests, since their access locations are closer to one another and would yield the best performance. Figure 8.3 shows the results of such an experiment on a Fujitsu drive. The spike, at just over 2 seconds, appears to be the close to or at the maximum request time able to be generated by the disk. Given this information, real-time applications that require a completion time greater than 2.03 seconds do not require anything out of the ordinary from the OS I/O scheduler. The on-disk scheduling will provide 40 MB/sec, while preventing starvation of real-time requests. Intriguingly, the spike is cleanly defined suggesting that the disk has a strong notion and control of completion times rather than simply counting requests before forcing a starved request to be served. Note that all disks do not have the same maximum starvation times. It does, however, appear likely that most disks do have some sort of built-in starvation mechanism, since not having one could lead to unexpectedly long delays and seemingly hung applications. On the disks that we have tested, the maximum starvation time ranges from approximately 1.5 to 2.5 seconds.

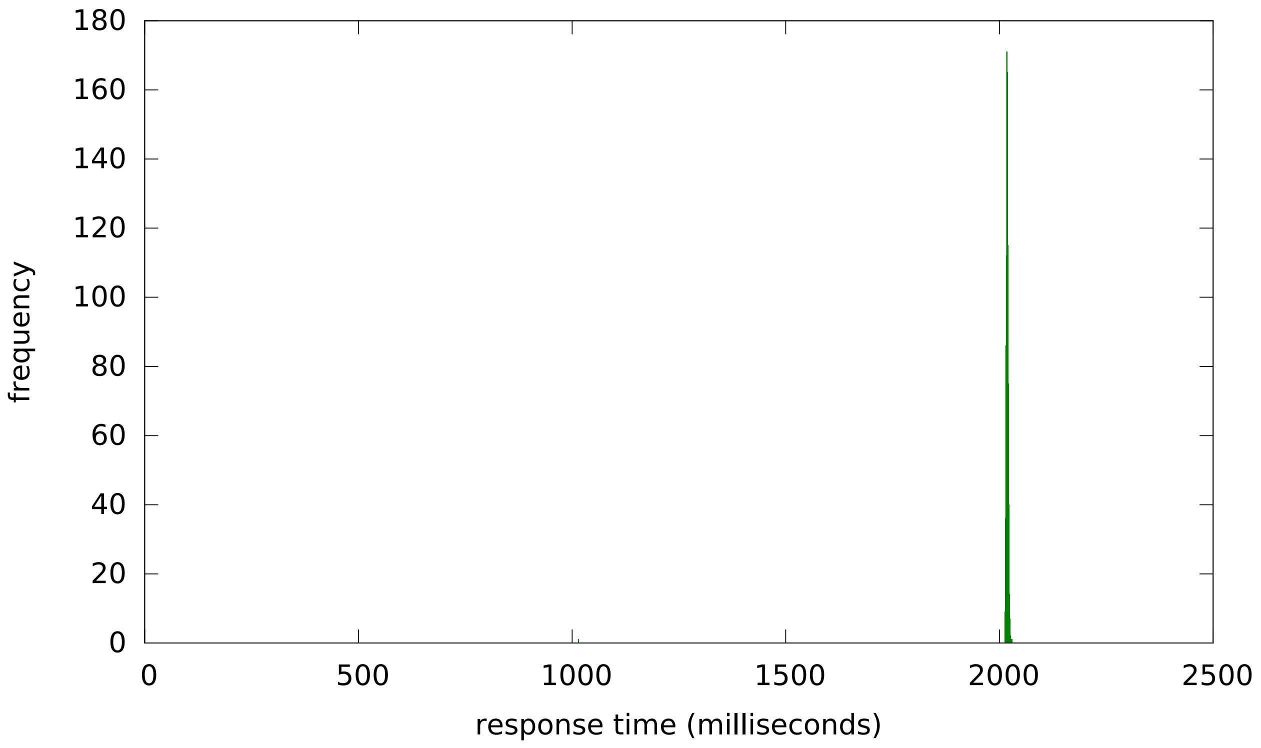


Figure 8.3: Disk's observed starvation prevention.

Should a real-time application require lower completion time than the disk-provided guaranteed completion time, additional mechanisms are needed.

8.3.2 “Draining" the On-Disk Queue

The reordering of the set of requests queued on disk when a real-time I/O request is issued is not the sole cause of the extended completion times in Figure 8.2. As on-disk queue slots become available, newly arrived requests can potentially be served before previously issued requests, leading to completion times greater than that of only servicing the number of possible requests permitted to be queued on a disk.

To determine the extent of starvation due to continuous arrivals of new requests, we first flooded the on-disk queue with 20 best-effort requests, then measured the time it takes for a real-time request to complete without sending further requests to the disk. As anticipated, the completion times are significantly shorter, as shown in Figure 8.4. Contrary to our intuition, real-time requests are more likely to be served with a shorter completion time, while a real-time request should have had an equal chance of being chosen among all the requests to be the next request to be served. With an in-house simulated disk, we realized that with command queuing, the completion time reflects both the probability of the real-time request being chosen as the nth request to be served, as well as the probability of various requests being coalesced to be served with fewer rotations. In this case, the probability of serving a real-time request as the last request while taking all twenty rotations is rather unlikely.

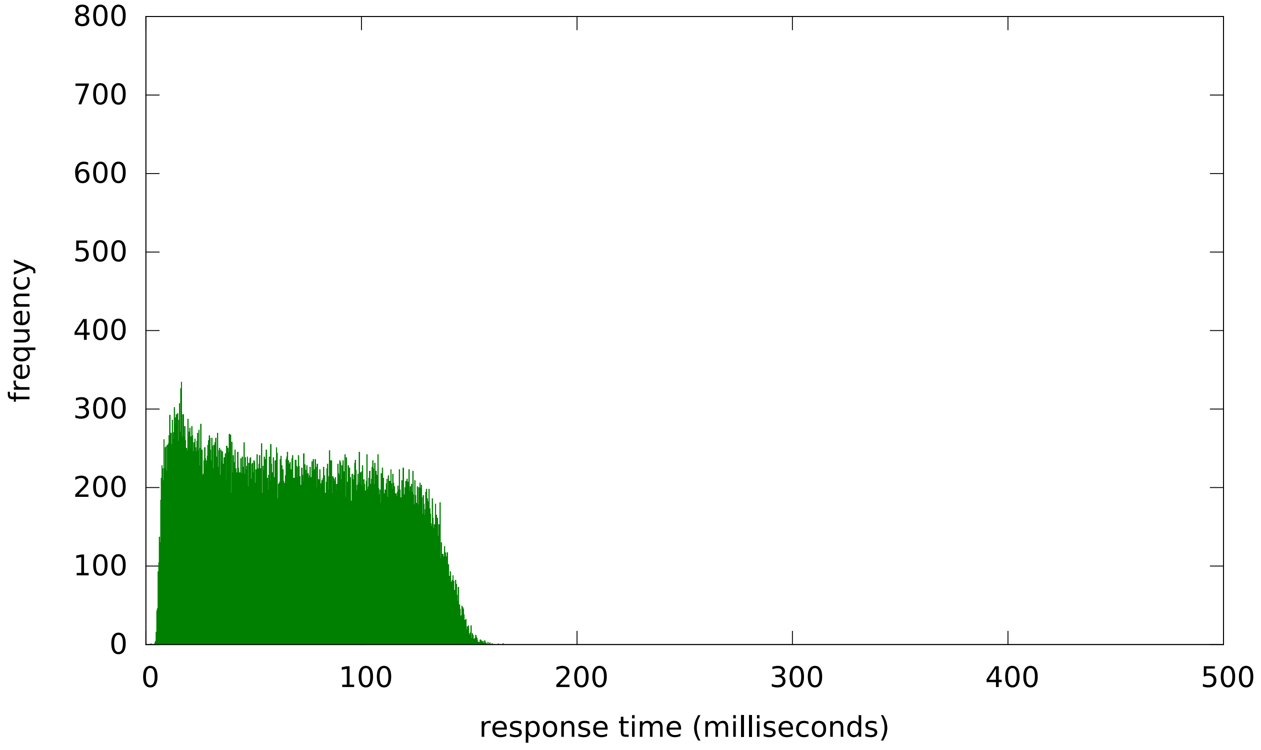


Figure 8.4: Effect of draining requests on the Fujitsu hard disk.

The “draining" mechanism used for this experiment provides a new means to bound completion times. Draining gives the disk a bounded, fewer number of choices to make when deciding the next request. As each best-effort request returns, it is increasing likely that the disk will serve the real-time request. However, in the worst-case, it is still possible that the real-time request will be served last.

While draining can prevent completion time constraints from being violated, it relies on *a priori* knowledge of the worst-case drain time for a given number of outstanding requests queued on the disk. Predicting this time can be difficult. One approach is to deduce the maximum possible seek and rotation latencies, based on possible service orderings for a given number of requests. However, the built-in disk scheduler comes preloaded in the firmware, with the overwhelming majority having undisclosed scheduling algorithms. Also, our observations show that on-disk scheduling exhibits a mixture of heuristics to prevent starvation. To illustrate, Figures 8.4, 8.5, and 8.6 used the same draining experimental framework on disks from different vendors. Based on the diversity of completion-time distributions, the difficulty of determining the scheduling algorithm is evident. Furthermore, even if one drive's scheduling algorithm is discovered and modeled, this does not generalize well with the diversity and rapid evolution of hard drives.

Given the complexities of modern disk drives, simple and general analytical prediction of the drain time may not be realistic. However, the maximum drain time can be determined empirically with a relatively high confidence level. To obtain the drain time for a given number of requests x, an experiment can be performed by sending x reasonably large requests to the disk with a uniformly random distribution across the entire disk. The time for all requests to return is then measured and recorded. The graph of the experiment for the drain time of 20 256-KB requests on the Fujitsu drive is shown in Figure 8.7. Using such an experiment, allows one to provide a reasonable upper-bound for the completion time for a real-time request with the presence of 19 outstanding best-effort requests.

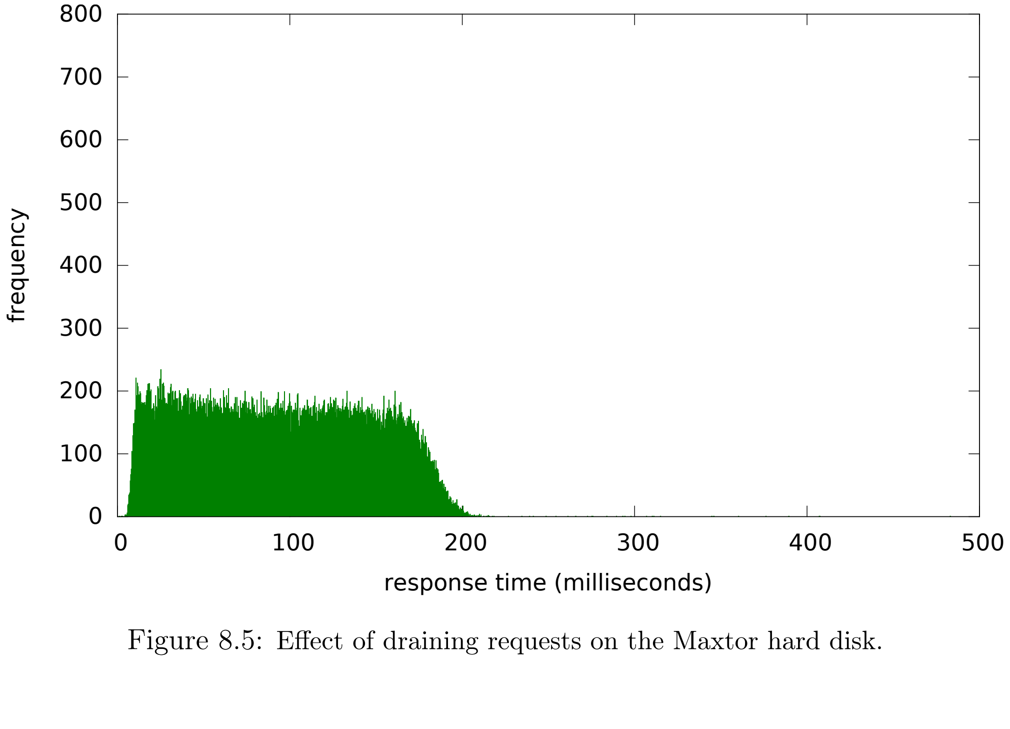


Figure 8.5: Effect of draining requests on the Maxtor hard disk.

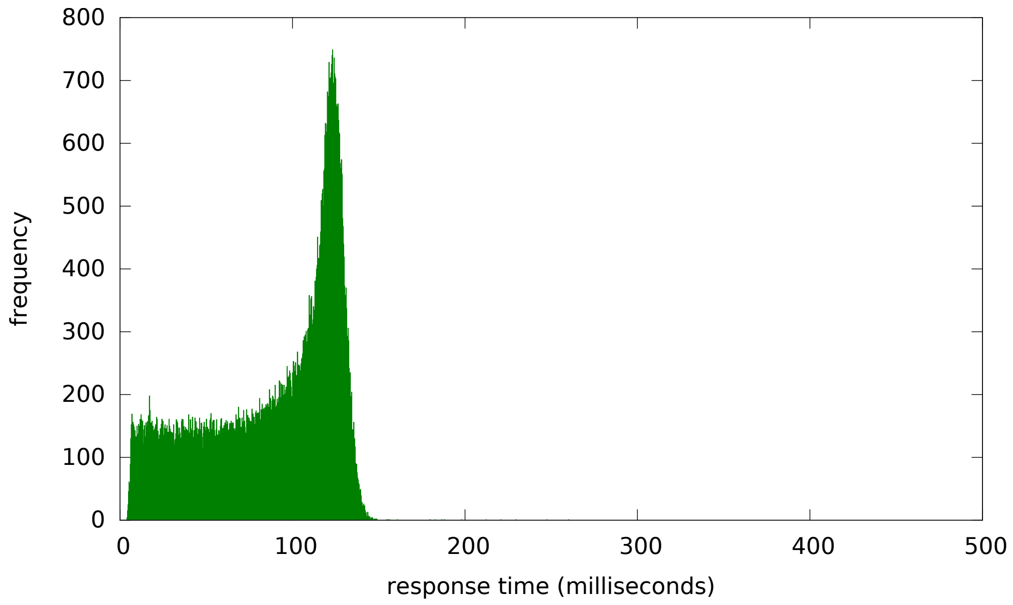


Figure 8.6: Effect of draining requests on the IBM hard disk.

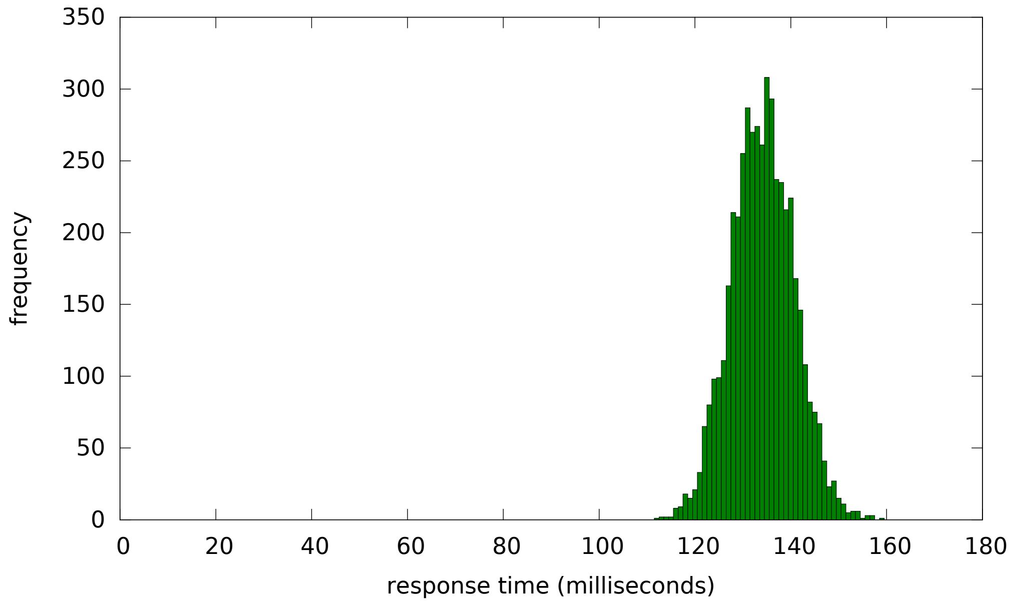


Figure 8.7: Empirically determining the drain time for 20 outstanding disk requests on the Fujitsu drive.

8.3.3 Experimental Verification

To implement the proposed draining policy, we modified our RTIO scheduler, so that once a real-time request has been issued to the disk, RTIO stops issuing further requests. If no real-time requests are present, RTIO limits the maximum number of on-disk best-effort requests to 19. For the experiment, two processes are used. One process is a periodic real-time task that reads 256-KB from a random disk location every 160 msec. The period is also considered to be the request's deadline. This deadline was chosen based on maximum drain time in Figure 8.7. The other process is a best-effort task that continuously issues 256-KB read requests with a maximum of 450 outstanding requests. Figure 8.8 shows the results of the experiment. As expected, no completion times exceeded 160 msec, meaning no deadlines were missed. The throughput remained at 40 MB/sec, suggesting that draining the entire queue occurred rather infrequently.

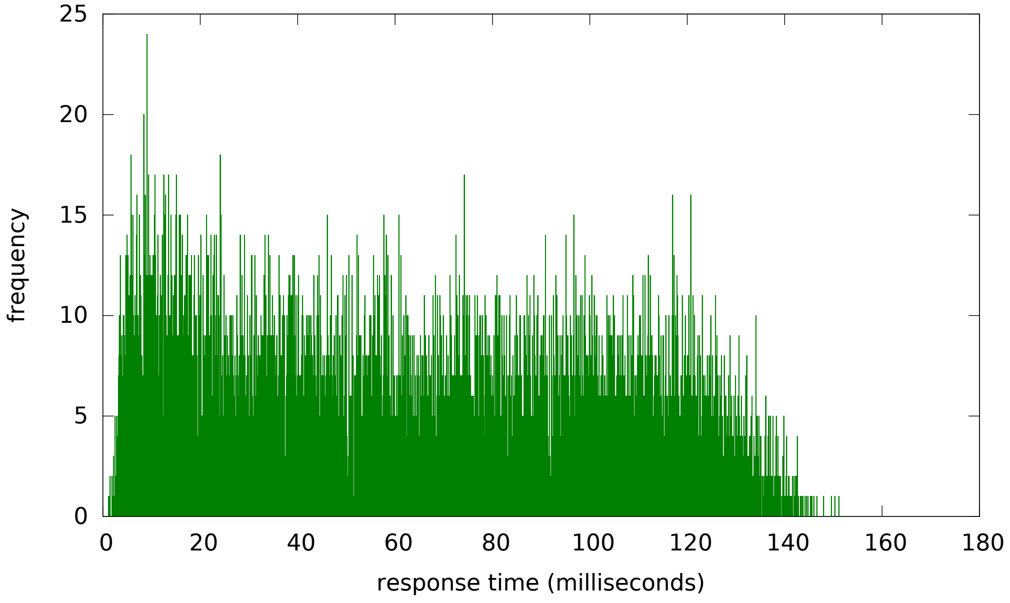


Figure 8.8: Draining the queue to preserve completion times of 160 msec.

8.3.4 Limiting the Effective Queue Depth

While draining helps meet one specific completion time constraint (e.g., 160 msec), configuring draining to guarantee arbitrary completion times (e.g., shorter deadlines) requires additional mechanisms. One possibility is to further limit the number of outstanding requests on disk, generalizing the draining technique. This mechanism effectively *artificially* limits the queue depth of the disk, thereby reducing maximum drain times. By determining and tabulating the drain time for various queue lengths (Figure 8.7), arbitrary completion time constraints can be met (subject to the time-related limitations of physical disks of course).

For instance, to meet the response time constraint of 75 msec, using a queue depth of less than 8 would suffice, from Figure 8.9. We would like to use the largest possible queue depth while still maintaining the desired completion time for real-time I/O requests. A larger queue length grants the disk more flexibility to create request service orders that result in higher throughput. Therefore, in this case, we would choose an on-disk queue depth of 7. The best-effort tasks must be limited to 6, with one slot reserved for the real-time request.

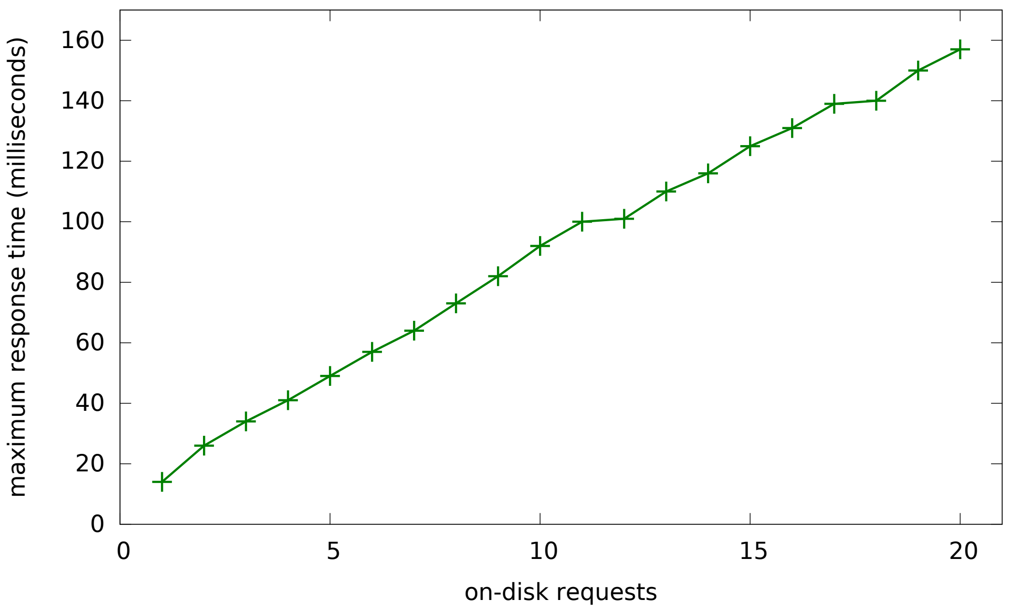


Figure 8.9: Maximum observed drain times for on-disk queue depths.

To verify timing constraints are met using the *a priori* tabulated queue length, a similar experiment was performed as before, changing only the period and deadline to be 75 msec. While Figure 8.10 shows that all deadlines are met as expected, we noticed that the throughput (not shown in the figure) for the best-effort requests dropped to 34 MB/sec. Interestingly, a 70% decline in queue length translates into only a 15% drop in bandwidth, demonstrating the effectiveness of on-disk queuing/scheduling even with a relatively limited queue depth.

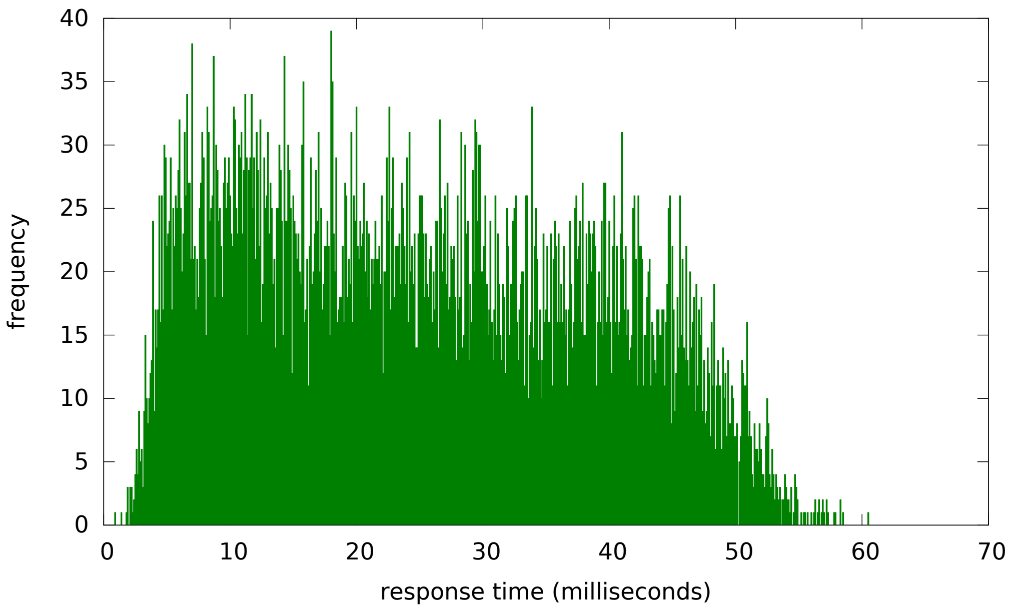


Figure 8.10: Limiting and draining the queue to preserve completion times of 75 msec.

8.4 Comparisons

To demonstrate the benefits of our approach to bound I/O completion times, we compared our RTIO scheduler with the default CFQ I/O scheduler for Linux. CFQ was chosen since it is the only standard Linux scheduler that uses I/O priorities when making scheduling decisions. Without this support, it is easy to see situations where a backlog of best-effort requests at the OS level may prevent a real-time request from reaching the disk in a timely manner, not to mention the requests queued on the disk. Given that the combined queue depth can be quite large, a real-time request may be forced to wait for hundreds of requests to be completed.

At first glance, the Linux *deadline* scheduler may appear to be effective in bounding completion times. However, the "deadline" in this case is the time before sending a request to the disk drive. Additionally, the deadline scheduler does not support the use of I/O priorities, meaning that all real-time I/Os will be handled the same as non-real-time I/Os. Figure 8.11 shows the results of using the Linux deadline scheduler with the workload generated through constantly sending 450 best-effort requests to the disk while at the same time recording the completion time of a single real-time request being issued periodically.

While the deadline scheduler could be modified to take advantage of the I/O priorities, the reordering problem of the disk's internal scheduler still exists. The problem is not entirely with getting the requests to the disk in a timely fashion, it is also getting them served by the disk within a specified deadline. The deadline scheduler could take advantage of "draining" by setting a deadline on requests that have yet to return from the disk as well as requests residing in the I/O scheduler's queue. When a request has been on the disk longer than a specified time, the draining mechanism should then be invoked. Draining would then last until no requests submitted to the disk have exceeded their deadlines. While this approach would not provide exact completion times for any given request, it would allow for much better time constrained performance than currently experienced.

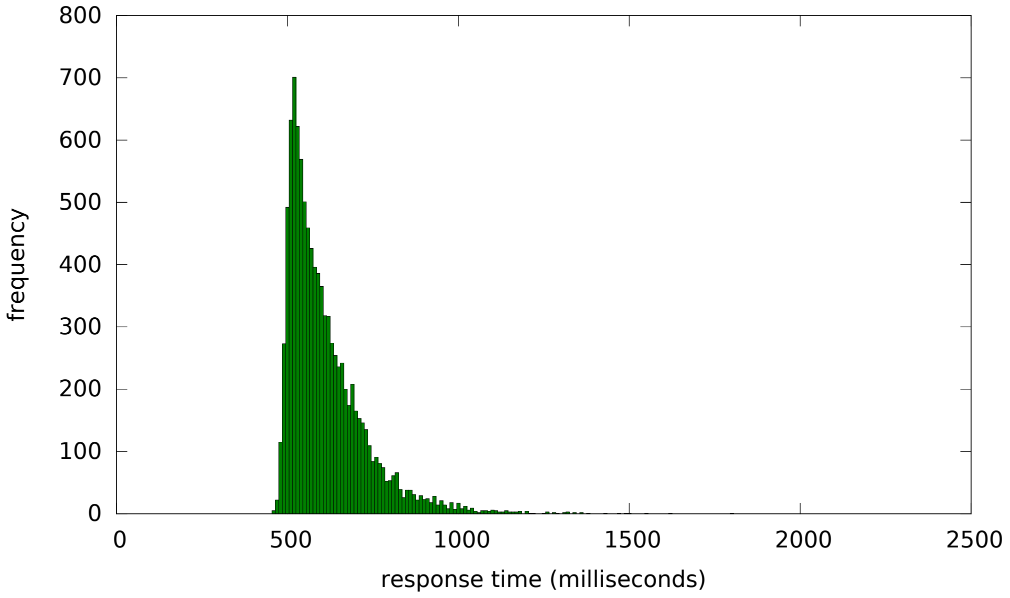


Figure 8.11: Completion times for real-time requests using the Linux deadline I/O scheduler.

To get an idea of the worst-case completion times of real-time requests sent to the disk using the CFQ scheduler, we repeated similar experiments to those in Figure 8.2. These experiments provided a continuous backlog of 450 best-effort requests to random locations on disk, in addition to one real-time request sent periodically. The request size was again limited to 256KB. The only difference was in the use of CFQ rather than RTIO.

Figure 8.12 shows that real-time completion times can exceed 2 seconds; however, at closer inspection, it appears that the real-time request handling is not correctly being implemented. When a new real-time request arrives, the CFQ scheduler does not always allow the newly issued real-time request to preempt preexisting requests in the driver's queue. In an attempt to alleviate this problem, we made a minor modification to the code to force the newly-issued real-time request to preempt all best-effort requests residing in the driver queue. This change, however, had the side effect of allowing only one request on the disk at a time. Although this change yielded low completions times with a maximum of 17 msec, the throughput degraded to 18 MB/sec.

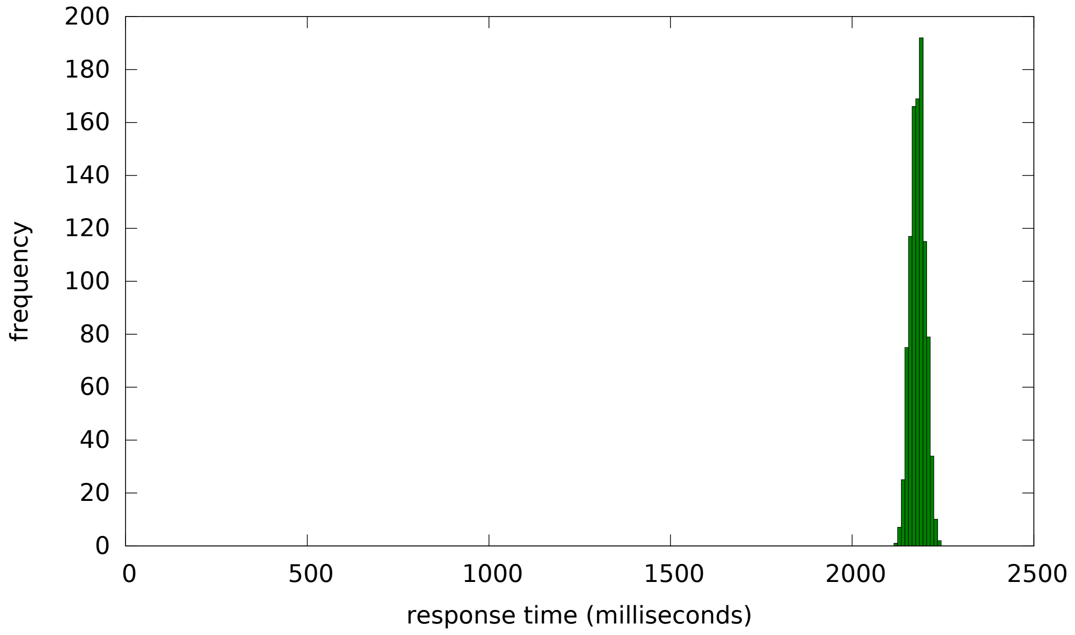


Figure 8.12: Completion times for real-time requests using the Linux CFQ I/O scheduler.

Further study and subsequently understanding of the CFQ code showed that the low throughput may be attributed to how CFQ treats each thread as a first-class scheduling entity. Because our experiments used numerous threads performing blocking I/Os, and since CFQ introduces artificial delays and anticipates that each thread will issue additional I/Os near the current requests location, our multi-threaded best-effort process can no longer result in multiple requests being sent to the on-disk queue. Asynchronous requests could have been used; however, the current GNU C library only emulates the asynchronous mechanism by spawning worker threads that subsequently perform blocking requests. Further, the maximum number of worker threads for emulating asynchronous requests is limited to 20. Our final solution to provide a more fair comparison with CFQ was to modify the CFQ code to schedule according to two groups of requests: one real-time and one best-effort. This modified CFQ now can forward multiple requests to the on-disk queue.

After making the modifications, the real-time completion times are shown in Figure 8.13. Without the limit imposed by the disk's starvation control algorithm, the real-time response might have been even worse. The real-time performance is poor because CFQ continuously sends best-effort requests to the disk, even though there may be real-time requests waiting to be served on the disk. Since the disk does not discern real-time and best-effort requests once they are in the on-disk queue, many best-effort requests can be served before serving the real-time request. At first it may seem peculiar that the majority of requests are over 2 seconds, considering the requests are randomly distributed across the entire disk, and real-time requests should have a good chance of not being chosen last. This problem occurs because CFQ sorts the random collection of best-effort requests, resulting in a stream of arriving requests that are more likely to be closer to the current disk head location than to the real-time request. Therefore, servicing the best-effort requests prior to the real-time request is more efficient. Combined with merging and sorting performed, CFQ issues near-sequential reads and can achieve throughput of 51 MB/sec. However, the cost is severe degradation of real-time performance.

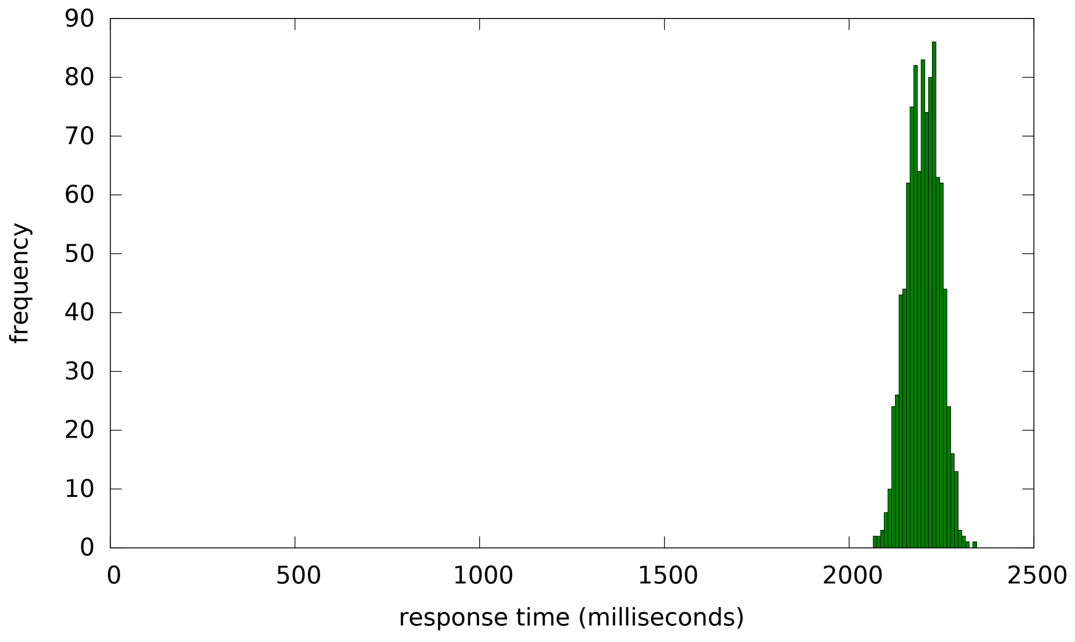


Figure 8.13: Completion times for real-time requests using a modified Linux CFQ I/O scheduler.

8.5 Conclusion

This chapter introduced the novel "draining" technique which is designed to leverage a hard disk's internal queue and scheduler without jeopardizing response time constraints for real-time requests.

While the same performance could theoretically be achieved through solely scheduling disk I/O requests from the operating system, allowing the disk to make scheduling decisions offloads the processor time necessary to schedule and achieves optimizations that cannot be easily (or practically) duplicated in the operating system. The approaches explored allow a disk to perform the work with its intimate knowledge of low-level hardware and physical constraints. Therefore, the request scheduling can be based on informed access to its near-future request information while reordering requests to realize efficient use of the disk's resource. Additionally, the disk can achieve a higher level of concurrency with CPU processing, servicing on-disk requests without immediate attention from the operating system. These approaches further allow high-priority real-time processes to use the CPU with little impact on disk performance.

The following chapter addresses the major limitation of the approaches presented in this chapter, which is that only one real-time request can be outstanding at one time. The limitation of one real-time request at a time effectively limits the support of only one real-time application by the system. At first glance, it may seem that the draining approach is easily extended to allow multiple real-time tasks. For instance one could reason that reserving multiple slots on the disk queue could allow for multiple real-time requests. However, such a solution may not work in many cases since the draining technique assumes that interference for a given I/O request can be limited. However, with multiple real-time requests, a given request may be starved by other real-time requests preempting it. This starvation could then lead to missed deadlines. In order to extend the draining approach, the analysis must consider the effect of interference not only from the non-real-time requests, but also from other real-time requests that could introduce additional amounts of priority inversion.

CHAPTER 9

DUAL-QUEUE FIFO: RT GUARANTEES FOR MULTIPLE TASKS

9.1 Introduction

This chapter presents a novel modification to FIFO scheduling, termed *dual-queue FIFO* (DQF), which extends the previous chapter's results. The results presented in this chapter provide timing guarantees for multiple sporadic tasks, rather than just a single sporadic task. DQF is designed to appropriately leverage the throughput performance of a hard disk's built-in scheduler in order to increase the number of supportable real-time task sets (i.e., guarantee more deadlines). By adjusting the amount of influence a built-in hard disk scheduler has on the service order of requests, higher levels of *guaranteed* throughput can be achieved, without jeopardizing deadlines.

Part of the difficulty for achieving useful, timely I/O service using hard disks is the separation of concerns and knowledge between a hard disk and the operating system (OS). Our assumption is that the OS has information about I/O request timing constraints (e.g., deadlines) and therefore orders requests based on some time-based metric. On the other hand, the hard disk firmware has intimate knowledge of the disk's operation and orders requests based on achieving high throughput, oblivious to timing constraints. Therefore, the problem is, how to tune (balance) the use of each scheduler in such a way as to meet the timing constraints of a given workload.

A key observation for understanding the necessity of using the hard disk's scheduler is that both sufficiently low latency and sufficiently high throughput are required to meet deadlines. Clearly, high latency can result in missed deadlines, but latency is also a function of the throughput. If the arrival rate of work is greater than the service rate for an arbitrarily large time interval, then deadlines will be missed. Decreasing the number of requests issued simultaneously to the disk tends to lower the worst-case latency at the disk level, but as a side effect also tends to lower the overall throughput.

Certainly one can imagine that intimate details of a hard disk can be extracted. Therefore, an OS scheduler can be created to take into account both I/O request time constraints and hard disk characteristics. However, the practicality of such an approach is questionable. First, there is a considerable hurdle establishing a detailed model of a given hard disk. Although the necessary information is typically stored in the disk's firmware, the firmware is typically not available in a form conducive to extracting the necessary information. Therefore, the general approach is to build a detailed model through numerous timing measurements. Such approaches tend to be very time consuming and the resulting model is often unique to a given disk. In fact, Krevat et al. [56] equate uniqueness of each hard disk to that of snowflakes. One of the many reasons for the difference in models is that the physical sector locations for a given logical block address (LBA) is often different amongst vendors and even models of the same vendor [89].

Another concern with making all I/O decisions from the OS is the computational and implementation complexity of the resulting scheduler. In general, the majority of general-purpose OS schedulers tend to use very simple online scheduling algorithms. For instance, one of the most well-known and used real-time scheduling algorithms is fixed-task priority, which determines the next task to run by choosing the task with the greatest numeric priority value (assuming a larger number represents a higher priority) out of the set of all ready tasks. Similarly, earliest-deadline-first sets the priority values based on deadlines, and on a per-job rather than per-task basis. There are many pragmatic reasons for the prevalence of simple scheduling algorithms, one of them being that general-purpose OSs tend to be extremely complex. Therefore, an implementation of a complex theoretical scheduling algorithm on an already complex system is likely to result in scheduling that deviates from the assumed theoretical model and thus potentially invalidates any theoretically predicted timing guarantees (e.g, see errors in the POSIX Sporadic Server described in Chapter 6). It should be noted that even though the online scheduling algorithm is simple that does not imply that offline timing analysis is also trivial.

The above considerations led us to explore what is achievable with a very simple on-line scheduling algorithm that leverages the hard disk built-in scheduler. The result is a scheduler we call dual-queue FIFO. The term dual-queue originates from the observation that hard disk I/O requests typically traverse through two queues, one in the OS and one hosted on the hard disk drive itself as discussed previously in Chapter 8. DQF operates by sending batches of requests to the hard disk *only* when it is idle and limits the number of requests issued to a predetermined maximum number. The intuition is that sending batches of requests improves throughput, but limiting the maximum number sent at once bounds worst-case disk service time (maximum sized non-preemptible time interval). The restriction on sending requests only when the disk becomes idle increases the tendency for batches of I/O requests to form under heavy load. Further, under light load, often only a small number of requests will be batched, thereby reducing the average-case response time.

This chapter first presents theoretical results for an upper bound on response time in Section 9.2 followed by an experimental evaluation using an implementation in Linux in Section 9.3.4. Related work is discussed in Section 9.4 followed by conclusions in Section 9.5.

9.2 Upper-Bound on Response Time

This section derives a theoretical upper bound on response time for a set of sporadic I/O tasks scheduled according to the DQF scheduling policy. The derived upper bound on response time is for an arbitrary job, which trivially extends to verifying the completion of I/O requests by a deadline.

9.2.1 Problem Definition

Given a sporadic I/O task set, T, scheduled according to the DQF scheduling policy, maximum disk queue size, K, and an amortized worst-case service time function, $b(i) \quad (i = 0, 1, ...K)$, calculate an upper bound on response time for any job of a given task.

9.2.2 Notation, Definitions, and System Model

Units of Work. I/O services provided by the system are described by the terms "request", "job", and "task". A *request* contains all the necessary information to dispatch a single I/O request to the hard disk (e.g., location, size, operation) with the constraint that the size of each of the I/O requests is less than or equal to some predefined maximum. A *job* defines a set of *requests* and a release time. The release time of a job denotes the earliest time instant all requests of that job are available to be dispatched to the disk. A task τ_i denotes a set of jobs such that the length of time between any two consecutive job release times has a minimum of p_i time units and each job has no more than e_i requests. Following the general convention, the term *sporadic task* is used to denote tasks whose period value represents a minimum on the inter-release time of consecutive jobs. The term periodic task is used for tasks whose period value represents an exact amount of time between consecutively released jobs. The *response time* of a given job is the length of time from its release to the time instant all of its I/O requests are served. The largest response time of all jobs in a task is said to be the task's response time. As one would expect, the response time of a task must be less than or equal to its deadline.

Time Representation. Intervals of time are expressed by two time instants designating the time interval's endpoints. Further, brackets and parentheses are used to denote whether or not the endpoints are included in a given time interval. As an example, $[t_1, t_2)$ represents the time interval consisting of all time instants t such that $\{t \mid t_1 \leq t < t_2\}$. The length of a time interval is defined to be $t_2 - t_1$, regardless of whether or not the endpoints are included in the time interval.

An intended distinction is made between the terms *maximum response time* and *upper bound on response time* when referring to a set of jobs. The maximum response time of a set of jobs is considered to be a response time value of at least one job in the given set of jobs. Comparatively, an upper bound on response time of a set of jobs need not have a corresponding job. An upper bound on response time only signifies that all job's response times are less than or equal to the upper bound on response time. The maximum response time is an upper bound on response time, but not necessarily vice versa.

Dual-Queue Scheduling Algorithm. The following details the notation and terminology used throughout this chapter to discuss dual-queue scheduling. An illustration of a sample I/O request's ($Request_k$) life cycle is shown in Figure 9.1 for reference. The sole focus of this chapter is the analysis of data I/O. Therefore, any use of the term *request* is assumed to mean *I/O request*.

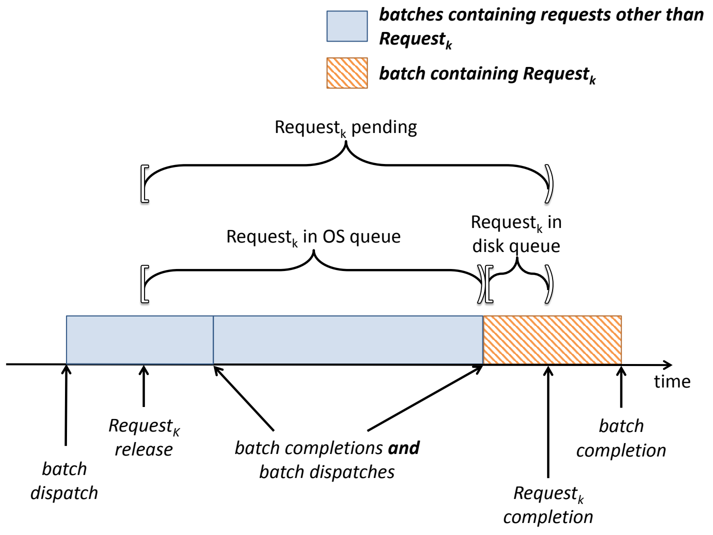


Figure 9.1: Life cycle of a given request denoted as $Request_k$.

The life of a request is considered to be the time interval it is pending in the system. A request is said to be *released* to the system the time instant it is available to be served by the disk and is detected by the system. A request is said to be *pending* at all time instants including the release time of the request up to, but not including, the instant its I/O operation is fully served (complete). Using time interval notation, if $t_{release}$ and $t_{complete}$ are the time instants a request is released and completed respectively, the request is pending in the time interval $[t_{release}, t_{complete})$.

Pending requests are either in the OS queue or the disk queue. The OS queue holds the subset of pending requests that have not been dispatched to the disk, while the disk queue holds the subset of requests dispatched to the disk but not served. A pending request must be in either the OS queue or the disk queue (not both). The set of pending requests in the system (those in either the OS or disk queue) at an arbitrary time instant t is denoted as $Q_{sys}(t)$. Similarly, those requests in the OS queue and the disk queue are denoted as $Q_{os}(t)$ and $Q_{disk}(t)$, respectively.

Subsets of pending requests are *dispatched* to the disk in *batches*. Only one batch is dispatched at a given time instant. Therefore, $B(t)$ is used to denote the unique batch dispatched at time instant t. A batch is said to *complete* the time instant all requests in the batch have been fully served by the disk. The number of requests in a batch is said to be the batch's *size* and may be any positive integer value up to and including some fixed upper bound, which we denote by z. The term $b(i)$ denotes the maximum duration required to serve a batch of size i and is assumed to be monotonically increasing function with respect to i defined for $i = 1, 2, \ldots, z$. The minimum time required to serve a batch is assumed to be non-zero.

The *OS I/O scheduler* dispatches batches of requests to the disk. The OS I/O scheduler is the only OS scheduler discussed in this chapter, so the abbreviated form *OS scheduler* is assumed to mean OS I/O scheduler. The OS scheduler is invoked at the time instant one or more requests are released and at the time instant a batch completes. The OS scheduler creates and dispatches a batch at any time instant it is invoked and the disk queue is empty. Therefore, no new requests are dispatched to the disk when the disk is busy serving a batch of requests.

The *OS scheduling policy* determines which requests are in a given batch. The scheduling policy is assumed to be work-conserving such that the number of requests in a batch dispatched at time instant t is $|B(t)| = min(|Q_{os}(t)|, z)$. That is, the OS scheduling policy will not intentionally idle the disk (by delaying the issue of requests to the disk) nor dispatch non-full batches if there are available requests. At this point in the discussion, the explicit scheduling policy (e.g., FIFO) is intentionally not yet specified so that notation and initial results are potentially applicable to any work-conserving scheduling policy.

The *on-disk scheduling policy* chooses the next request in the disk queue to service. As in the OS scheduling policy, the only assumption is that the policy is work conserving. Refinements to the on-disk scheduling policy will not be considered in this section since one of the main motivations for formulating dual-queue scheduling is to improve real-time I/O performance with the assumption that the order of requests served by the disk is unknown.

Capacity. The term *capacity* is used to denote the ratio of a task's *number of requests released* per period to the size of the period. Capacity is similar to processor *utilization*, but considers a number of requests rather than execution time in order to leverage worst-case service time amortization in schedulability analysis. The capacity of a task τ_i is denoted as u_i and is defined as:

$$u_i = \frac{e_i}{p_i} \tag{9.2.2.1}$$

where e_i is the maximum number of requests any job in τ_i can release and p_i is the period of τ_i.

U_{sum} denotes the capacity of a system task set and is defined as follows:

$$U_{sum} = \sum_{i=1}^{n} u_i \tag{9.2.2.2}$$

Intuitively, no bound on response time can be guaranteed if the capacity of a system task set is greater than the maximum a priori defined service rate, $(z/b(z))$. Therefore, the following bound on capacity is assumed for any considered task set:

$$U_{sum} \leq \frac{z}{b(z)} \tag{9.2.2.3}$$

9.2.3 Busy Interval

The response time of an arbitrary sporadic task τ_k is established by deriving an upper bound on the number of requests served in a *busy interval* of an arbitrary job J_k in τ_k. Many real-time processor scheduling analyses are based on the concept of a busy interval [11, 39, 59]. Generally speaking, a busy interval is a time interval when the resource under consideration has pending work. For work conserving scheduling algorithms, then the processing resource is continuously busy. Figure 9.2 illustrates our busy interval.

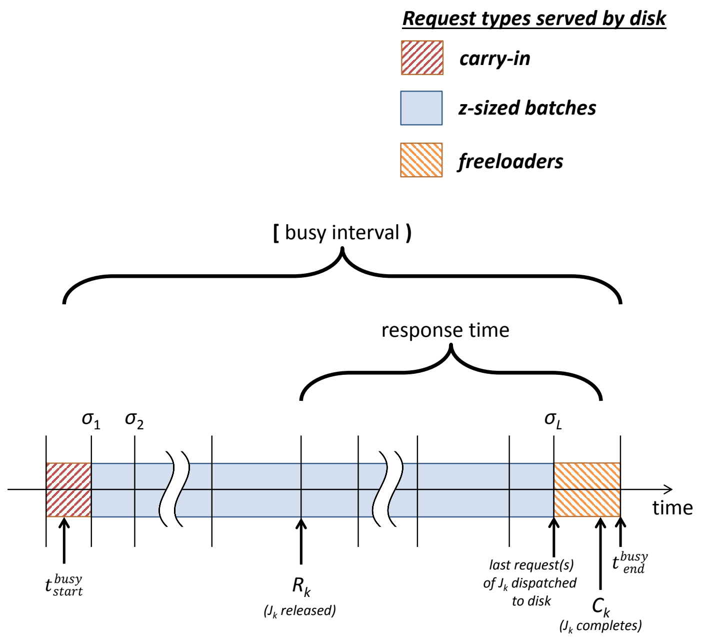


Figure 9.2: Illustration of J_k's busy interval.

The following describes the busy interval used in this chapter. The busy interval of J_k is denoted by the time interval $[t^{start}_{busy}, t^{end}_{busy})$ where t^{end}_{busy} is the completion time of the batch serving requests at time instant C_k. Put another way, t^{end}_{busy} is the time instant the batch containing the latest served request of J_k completes. t^{start}_{busy} is the latest time instant at or before R_k such that $|Q_{os}(t)|$ transitions from 0 to greater than 0. Since no requests are released before the system's start time and the disk takes some non-zero time to serve a batch, a valid t^{start}_{busy} and t^{end}_{busy} always exists. Each batch dispatched in the busy interval has a start time denoted by σ_i, where i is the one-based index of the i^{th} batch dispatched to the disk in a busy interval. The first batch is dispatched at time instant σ_1 and the final batch is dispatched at time instant σ_L where $(L = 1, 2, \ldots)$. It may be that $L = 1$. In general, $t^{start}_{busy} \leq \sigma_1 \leq \sigma_L < t^{end}_{busy}$.

A critical property that results from the particular choice of t^{start}_{busy} and t^{end}_{busy} is that all but the last batch dispatched in any busy interval must contain z requests.

Lemma 9.2.1

All but the last batch dispatched in a busy interval must contain z requests.

Proof.

Consider 2 cases: (a) $L = 1$ (b) $L > 1$.

(a) B_L is the last and only batch and the lemma therefore trivially true.

(b) Suppose a batch other than B_L, call it B_f, contains fewer than z requests. Such a batch's start time must be either in (1) $[\sigma_1, R_k)$, or (2) $[R_k, \sigma_L)$.

(1) Suppose the start time of B_f is in the time interval $[\sigma_1, R_k)$.

t_{busy}^{start} is chosen as the latest time instant at or before R_k where $Q_{os}(t)$ transitions from 0 to greater than 0 requests. If B_f is not full, then the time instant B_f is dispatched, $Q_{os}(\sigma_f)$ must be zero. Some later time instant t' must exist after σ_f when a job is put in the OS queue resulting in $Q_{os}(t) > 0$. Since J_k has yet to be put in the OS queue, such a job must always exist. t' is the start of the busy interval and after σ_f, contradicting the initial assumption.

(2) Suppose the start time of B_f is in the time interval $[R_k, \sigma_L)$.

The OS scheduling algorithm is assumed to be work conserving. Therefore, the OS scheduler must have dispatched requests of J_k to ensure all batches are full. If B_f is not full, either all requests of J_k are completing in B_f, meaning $B_f = B_L$ or J_k has already completed meaning B_f is outside the busy interval. Either contradicts the initial assumption.

$\square$

9.2.4 Summary of Notation

$\boldsymbol{\tau}$	A sporadic task denoting a set of jobs such that the inter-release of any two consecutive jobs is no less than the task's period attribute.
$\boldsymbol{b(i)}$	Maximum length of time required to serve a batch of size i.
$\boldsymbol{B(\sigma_i)}$	Set of requests in batch released at σ_i.
$\boldsymbol{B_i}$	Number of requests in batch released at σ_i. More precisely, $B_i = \lvert B(\sigma_i)\rvert$.
$\boldsymbol{[t_{busy}^{start}, t_{busy}^{end})}$	A busy interval.
$\boldsymbol{u_i}$	Capacity of task τ_i, equal to e_i/p_i.

$\boldsymbol{U_{sum}}(\mathbf{T})$	Capacity of system task set T.
$\boldsymbol{J_k}$	An arbitrary job..
$\boldsymbol{R_k}$	Time instant job J_k is released..
$\boldsymbol{C_k}$	Time instant job J_k is completely served..
$\boldsymbol{K}$	Maximum queue size that the disk can support.
$\boldsymbol{e_i}$	Maximum number of requests released by a single job of τ_i.
$\boldsymbol{p_i}$	Period of τ_i.
$\boldsymbol{p_{min}}(\mathbf{T})$	The smallest period of all tasks in T.
$\boldsymbol{Q_{disk}(t)}$	Length of the disk queue at time instant t.
$\boldsymbol{Q_{os}(t)}$	Length of the OS queue at time instant t.
$\boldsymbol{Q_{sys}(t)}$	Sum of the OS and disk queue lengths at time instant t.
$\boldsymbol{\sigma_l}$	Time instant batch the l^{th} batch of a given busy interval is dispatched to the disk.
$\mathbf{T}$	A set of sporadic tasks.
$\boldsymbol{S_{total}}$	Total number of requests both released and served in a given busy interval.
$\boldsymbol{z}$	Maximum number of requests in a batch as permitted by a given instance of the DQF scheduling algorithm.

9.2.5 Linear Upper Bound

The following derives an upper bound on response time for a dual queue scheduled system where the requests dispatched to a disk are ordered using the FIFO scheduling policy. This scheduling algorithm is termed DQF.

The analysis starts with an upper bound on an arbitrary job's response time by considering the number of requests released and served in a busy interval. The total number of requests both released and served in a given busy interval, represented by S_{total}, can be calculated by summing all batches released in the busy interval. Recall that a busy interval ends at the completion time of the batch containing the latest served request of J_k not the completion time of J_k. Therefore, all requests in $B(\sigma_L)$ are served in the busy interval. Further, any requests released after σ_L are not served in the busy interval since dual-queue scheduling only dispatches batches when the disk is idle. Since the batch released at σ_L is the last batch of the busy interval any subsequent requests released after σ_L cannot be part of J_k's busy interval. Additionally, note that carry-in requests are not included in S_{total}. While the requests in the carry-in batch may be served in the busy interval, such requests are released prior to t_{busy}^{start}. For notational convenience, let $B_i = |B(\sigma_i)|$. We will now derive an upper-bound on an arbitrary job's response time.

Lemma 9.2.2

The following inequality is an upper bound on the response time of an arbitrary job J_k released in a busy interval:

$$C_k - R_k \le \frac{b(z)}{z}\left(\sum_{i=1}^{n} e_i\right) + b(B_1) + b(B_L) - \frac{b(z)}{z} \tag{9.2.5.1}$$

Proof.

The sum of requests in all batches dispatched in a busy interval can be calculated as follows,

$$S_{total} = \sum_{i=1}^{L} B_i \tag{9.2.5.2}$$

Pulling out the last batch of the busy interval from the summation gives,

$$S_{total} = \sum_{i=1}^{L-1} B_i + B_L \tag{9.2.5.3}$$

Note for $L = 1$, the summation in the above equation results in an empty sum, which is considered to evaluate to 0.

Lemma 9.2.1 states that all batches dispatched in the busy interval except B_L contain z requests. So, replacing all batches in the summation with z gives,

$$S_{total} = (L-1)z + B_L \tag{9.2.5.4}$$

The requests *served* (not necessarily released) in a busy interval can be partitioned into three categories: the carry-in requests $S_{c_{in}}$ (released prior to but served in the busy interval see Section 9.2.5), the requests released in the busy interval at or prior to R_k, and the freeloaders S_{free} (requests released after R_k but served in the busy interval). The notation $Rel\Big(\textit{time interval}\Big)$ is used to denote the number of requests released in a specified time interval. As such, the total number of requests released and served in a busy interval can calculated as follows:

$$S_{total} = Rel\Big([t_{busy}^{start}, R_k]\Big) + S_{free} \tag{9.2.5.5}$$

Note that J_k is released at R_k and is counted in the term $Rel\Big([t_{busy}^{start}, R_k]\Big)$.

The periodic constraint of a sporadic task upper bounds the number of requests released in a Δ length of time to be no greater than $\sum_{i=1}^{n} \left\lceil \frac{\Delta}{p_i} \right\rceil e_i$. Therefore, substituting $Rel\Big([t_{busy}^{start}, R_k]\Big)$ by such an upper bound gives,

$$S_{total}^{FIFO} \leq \left(\sum_{i=1}^{n} \left\lceil \frac{R_k - t_{busy}^{start}}{p_i} \right\rceil \cdot e_i \right) + S_{free} \tag{9.2.5.6}$$

Substituting S_{total} in the above equation with the RHS of Equation 9.2.5.4 gives,

$$(L-1) \cdot z + B_L \leq \sum_{i=1}^{n} \left(\left\lceil \frac{R_k - t_{busy}^{start}}{p_i} \right\rceil \cdot e_i \right) + S_{free} \tag{9.2.5.7}$$

The final batch of a busy interval B_L must include at least one request from J_k by definition, so $B_L \geq S_{free} + 1$. Solving for S_{free} results in $S_{free} \leq B_L - 1$. Substituting $B_L - 1$ for S_{free} in the above equation,

$$(L-1) \cdot z + B_L \leq \sum_{i=1}^{n} \left(\left\lceil \frac{R_k - t_{busy}^{start}}{p_i} \right\rceil \cdot e_i \right) + B_L - 1 \tag{9.2.5.8}$$

Eliminating the common B_L terms from both sides of the inequality,

$$(L-1) \cdot z \leq \sum_{i=1}^{n} \left(\left\lceil \frac{R_k - t_{busy}^{start}}{p_i} \right\rceil \cdot e_i \right) - 1 \tag{9.2.5.9}$$

An upper-bound on the length of a busy interval is the sum of the maximum time to serve the carry-in requests and all requests released and served in the busy interval. Lemma 9.2.1 states that all but the last batch dispatched in the busy interval must contain z requests. The first batch released in the busy is delayed by the carry-in batch. Since $b(B_i)$ denotes the maximum time to serve B_i requests an upper-bound is calculated as follows,

$$t_{busy}^{end} - t_{busy}^{start} \leq b(S_{c_{in}}) + (L-1) \cdot b(z) + b(B_L) \tag{9.2.5.10}$$

Solving for $L - 1$ gives,

$$L - 1 \geq \frac{t_{busy}^{end} - t_{busy}^{start} - b(S_{c_{in}}) - b(B_L)}{b(z)} \tag{9.2.5.11}$$

Substituting $L-1$ in Equation 9.2.5.9 with the right-hand side of the above inequality gives,

$$\frac{t_{busy}^{end} - t_{busy}^{start} - b(S_{c_{in}}) - b(B_L)}{b(z)} \cdot z \leq \sum_{i=1}^{n} \left(\left\lceil \frac{R_k - t_{busy}^{start}}{p_i} \right\rceil \cdot e_i \right) - 1 \tag{9.2.5.12}$$

Removing the ceiling using the inequality relationship, $\lceil x \rceil \leq x+1$, gives

$$\frac{t_{busy}^{end} - t_{busy}^{start} - b(B_1) - b(B_L)}{b(z)} \cdot z \leq \sum_{i=1}^{n} \left((\frac{R_k - t_{busy}^{start}}{p_i} + 1) \cdot e_i \right) - 1 \tag{9.2.5.13}$$

Distributing e_i,

$$\frac{t_{busy}^{end} - t_{busy}^{start} - b(B_1) - b(B_L)}{b(z)} \cdot z \leq \sum_{i=1}^{n} \left(\frac{(R_k - t_{busy}^{start}) \cdot e_i}{p_i} + e_i \right) - 1 \tag{9.2.5.14}$$

Distributing the summation over addition and factoring out $R_k - t_{busy}^{start}$ gives,

$$\frac{t_{busy}^{end} - t_{busy}^{start} - b(B_1) - b(B_L)}{b(z)} \cdot z \leq \left(R_k - t_{busy}^{start}\right) \left(\sum_{i=1}^{n} \frac{e_i}{p_i} \right) + \left(\sum_{i=1}^{n} e_i \right) - 1 \tag{9.2.5.15}$$

The system is assumed to not be permanently overloaded, so $\frac{z}{b(z)} \geq \sum_{i=1}^{n} \frac{e_i}{p_i}$ (Equation 9.2.2.3),

$$\frac{t_{busy}^{end} - t_{busy}^{start} - b(B_1) - b(B_L)}{b(z)} \cdot z \leq \left(R_k - t_{busy}^{start}\right) \frac{z}{b(z)} + \left(\sum_{i=1}^{n} e_i \right) - 1 \tag{9.2.5.16}$$

Multiplying both sides of the above inequality by $\frac{b(z)}{z}$,

$$t_{busy}^{end} - t_{busy}^{start} - b(B_1) - b(B_L) \leq \left(R_k - t_{busy}^{start}\right) + \frac{b(z)}{z} \left(\sum_{i=1}^{n} e_i \right) - \frac{b(z)}{z} \tag{9.2.5.17}$$

The completion time of J_k is C_k. From the definition of a busy interval, J_k must complete at or before the end of the busy interval, so $C_k \leq t_{busy}^{end}$. Substituting t_{busy}^{end} by C_k gives,

$$C_k - t_{busy}^{start} - b(B_1) - b(B_L) \leq \left(R_k - t_{busy}^{start}\right) + \frac{b(z)}{z} \left(\sum_{i=1}^{n} e_i \right) - \frac{b(z)}{z} \tag{9.2.5.18}$$

Eliminating common t_{busy}^{start} terms gives,

$$C_k - b(B_1) - b(B_L) \leq R_k + \frac{b(z)}{z} \left(\sum_{i=1}^{n} e_i \right) - \frac{b(z)}{z} \tag{9.2.5.19}$$

Solving for $C_k - R_k$, the response time of J_k, gives

$$C_k - R_k \leq \frac{b(z)}{z} \left(\sum_{i=1}^{n} e_i \right) + b(B_1) + b(B_L) - \frac{b(z)}{z} \tag{9.2.5.20}$$

□

A sanity check for validating the correctness of the resulting bound is to verify that the resulting units are as expected. Both sides of Equation 9.2.5.20 should have units of time. Recall that the units of e_i are fixed-sized data requests (not time, as with conventional processor scheduling), therefore, the units on the right hand side of Equation 9.2.5.20 is time, which matches the expectation.

A perceived limitation of the above upper bound is that it only applies to jobs released in a busy interval. However, the following shows that all jobs must be released in a busy interval and therefore such a limitation does not meaningfully affect the derived upper bound.

Theorem 9.2.3

Equation 9.2.5.20 is a valid upper bound for any job.

Proof.

Equation 9.2.5.20 provides an upper bound for an arbitrary job in a busy interval. Therefore, the following shows that a valid busy interval exists for any job.
A busy interval is defined by the time instants t_{busy}^{start} and t_{busy}^{end} of an arbitrary job J_k. The system is assumed to not be overloaded, so the long term service rate is sufficient to serve the total number of requests released. As such, each job must eventually complete and so t_{busy}^{end} must exist. Since no requests are released prior to the system start time, Q_{os} before the system start time is 0. Q_{os} cannot be negative and the release of J_k increases Q_{os} by at least one request. This means that Q_{os} must transition at least once at or before R_k. Therefore, a valid t_{busy}^{start} must always exist. □

The above upper-bound on response time can be described in terms of supply and request bound functions used for network flow analysis [58] and later for processor analyses described in [19, 39]. A request bound function, $rbf(\Delta)$, is an upper bound on the cumulative number of requests that may be released within any time interval of length Δ. A supply bound function, $sbf(\Delta)$, is a lower bound on the cumulative service capacity provided by the disk in any time interval of length Δ.

Intuitively, Equation 9.2.5.20 is the maximum horizontal distance between any two points on the sbf and rbf. That is, Figure 9.3 illustrates examples of sbf and rbf curves.

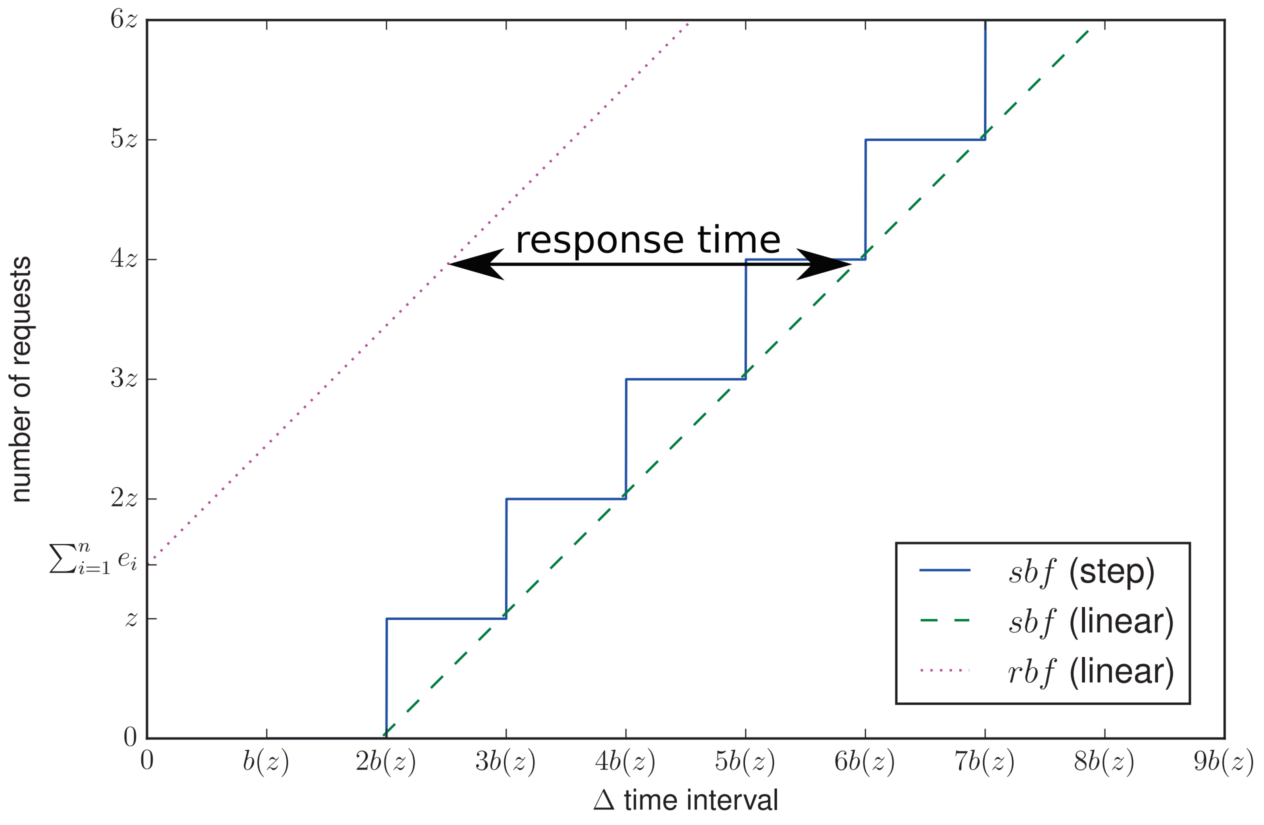


Figure 9.3: Request and supply curves for illustrating the maximum response time.

The linear rbf is derived using the periodic constraint. Recall that the periodic constraint of a sporadic task upper bounds the number of requests released in a Δ length of time to be no greater than $\sum_{i=1}^{n} \left\lceil \frac{\Delta}{p_i} \right\rceil e_i$.

$$rbf(\Delta) \leq \sum_{i=1}^{n} \left\lceil \frac{\Delta}{p_i} \right\rceil e_i \tag{9.2.5.21}$$

Removing the ceiling using the inequality relationship, $\lceil x \rceil \leq x + 1$ gives,

$$rbf(\Delta) \leq \sum_{i=1}^{n} \left(\frac{\Delta}{p_i} + 1 \right) e_i \tag{9.2.5.22}$$

Distributing e_i gives,

$$rbf(\Delta) \leq \sum_{i=1}^{n} \left(\Delta \cdot \frac{e_i}{p_i} + e_i \right) \tag{9.2.5.23}$$

Distributing the summation over addition gives

$$rbf(\Delta) \leq \sum_{i=1}^{n} \frac{e_i}{p_i} \cdot \Delta + \sum_{i=1}^{n} e_i \tag{9.2.5.24}$$

The system is assumed to not be permanently overloaded, so $\frac{z}{b(z)} \geq \sum_{i=1}^{n} \frac{e_i}{p_i}$ (Equation 9.2.2.3),

$$rbf(\Delta) \leq \frac{z}{b(z)} \cdot \Delta + \sum_{i=1}^{n} e_i \tag{9.2.5.25}$$

The *sbf* for any request released in a busy interval can be derived from the Lemma 9.2.1, which states that all but the latest batch dispatched in a busy interval must contain exactly z requests. Further, recall that the carry-in batch cannot exceed $b(z)$ and S_{free} cannot exceed $z-1$. Therefore, z requests must be served every $b(z)$ time units following an initial delay of $2 \cdot b(z) - \frac{b(z)}{z}$. This minimum service can be expressed as follows,

$$sbf\,(\Delta) \geq \begin{cases} \left\lceil \frac{z}{b(z)} \right\rceil \Delta - 2z, & \text{if } \Delta \geq 2b(z) \\ 0, & \text{otherwise.} \end{cases} \tag{9.2.5.26}$$

Removing the ceiling function gives,

$$sbf\,(\Delta) \geq \begin{cases} \frac{z}{b(z)}\Delta - 2z, & \text{if } \Delta \geq 2b(z) \\ 0, & \text{otherwise.} \end{cases} \tag{9.2.5.27}$$

which is the linear *sbf* in Figure 9.3.

The derived linear bound includes a negative term that at first may seem counterintuitive. This term is attributed to removing one request of J_k that would otherwise be counted twice. The *sbf* curve effectively provides an equivalent view the actual service sequence that occurs on the system. The *sbf* shows that the actual service rate of requests at a rate of $\frac{z}{b(z)}$ after a delay to serve the first and last batches is no different than delaying for the carry-in serving requests and finally delaying for any carry-in. At least one request of J_k must be served in the last batch from the definition of the busy interval. All requests of the backlog and J_k are effectively served at a rate of $\frac{z}{b(z)}$ after the delay of the first and last batches. Since the last batch must have 1 requests of J_k the effect on the linear *sbf* is that 1 request must have been served prior to the $\frac{z}{b(z)}$ service rate.

9.2.6 Tighter Upper Bound

The following describes a tighter upper bound on response time by leveraging the knowledge that a sufficient number of requests must be released to achieve longer busy intervals.

Recall from Lemma 9.2.1 that each batch dispatched in a busy interval, except the last, must be full. Therefore, at the dispatch of each subsequent batch the cumulative number of requests released since the start of the busy interval must be greater than z times the number of dispatched batches, otherwise, the busy interval would end at the completion of the newly dispatched batch. Therefore, a lower-bound on the number of requests released can be determined from the assumption that a batch can take no longer than $b(z)$ time to complete.

The following lemma provides a means to calculate an upper bound on L.

Lemma 9.2.4

Any positive integer x that satisfies the inequality,

$$\sum_{i=1}^{n} \left\lceil \frac{b(S_{c_{in}}) + (x-1)\cdot b(z)}{p_i} \right\rceil e_i \leq x \cdot z$$

is a valid upper-bound on L (i.e., $L \leq x$).

Proof.

Lemma 9.2.1 states that all but the last batch dispatched at σ_L contain z requests. Further, following from the definition of a busy interval, $Q_{os}(\sigma_x) > z$ for all batch dispatch time instants $\sigma_1, \sigma_2, \ldots, \sigma_{L-1}$ (otherwise the length of the OS queue would have been 0). Therefore, the number of requests released in the time interval $[t_{busy}^{start}, \sigma_x]$ for any $x \leq L-1$ must be greater than $x \cdot z$. Any value of x that does not satisfy this constraint on the length of the OS queue must be σ_L or a later batch release time. Expressing the requests released at each σ_x as a sequence gives

$$\begin{aligned}
Rel\Big([t_{busy}^{start}, \sigma_1]\Big) &> 1 \cdot z \\
Rel\Big([t_{busy}^{start}, \sigma_2]\Big) &> 2 \cdot z \\
&\vdots \\
Rel\Big([t_{busy}^{start}, \sigma_x]\Big) &> x \cdot z \qquad \forall x : 1 \leq x < L
\end{aligned} \tag{9.2.6.1}$$

Replacing the number of requests released by the upper bound due to the periodic constraint gives

$$\sum_{i=1}^{n} \left\lceil \frac{\sigma_x - t_{busy}^{start}}{p_i} \right\rceil e_i > x \cdot z \qquad \forall x : 1 \leq x < L \tag{9.2.6.2}$$

σ_x can be no greater than the maximum time to serve $S_{c_{in}}$ requests plus the z-sized batches,

$$\begin{aligned}
\sigma_1 &\leq t_{busy}^{start} + b(S_{c_{in}}) + (1-1)\cdot b(z) \\
\sigma_2 &\leq t_{busy}^{start} + b(S_{c_{in}}) + (2-1)\cdot b(z) \\
&\vdots \\
\sigma_x &\leq t_{busy}^{start} + b(S_{c_{in}}) + (x-1)\cdot b(z) \qquad \forall x : 1 \leq x < L
\end{aligned} \tag{9.2.6.3}$$

Solving the above inequality for $\sigma_x - t_{busy}^{start}$ gives

$$\sigma_x - t_{busy}^{start} \leq b(S_{c_{in}}) + (x-1) \cdot b(z) \qquad \forall x : 1 \leq x < L \tag{9.2.6.4}$$

Substituting $\sigma_x - t_{busy}^{start}$ in 9.2.6.2 with the RHS of the above inequality gives

$$\sum_{i=1}^{n} \left\lceil \frac{b(S_{c_{in}}) + (x-1) \cdot b(z)}{p_i} \right\rceil e_i > x \cdot z \qquad \forall x : 1 \leq x < L \tag{9.2.6.5}$$

σ_L is the earliest time instant at or after σ_1 when $Q_{os} \leq z$. It follows then that if $1 \leq x < L$ then the above inequality is satisfied. We will now take the contrapositive.
Therefore, the negation of the above inequality,

$$\sum_{i=1}^{n} \left\lceil \frac{b(S_{c_{in}}) + (x-1) \cdot b(z)}{p_i} \right\rceil e_i \leq x \cdot z \tag{9.2.6.6}$$

implies $L \leq x$. □

L **Iteration Upper-Bound.** Finding an upper bound on L using the above Lemma 9.2.4 allows us to calculate an upper bound on any job's response time by testing successive integer values of x starting with $x = 1$. We denote the smallest value of x (if any) that satisfies this test by $\hat{L}$. $\hat{L}$ is an upper bound on the number of batches *dispatched* in a busy interval. At most one additional batch may be in progress at the start of the busy interval. Further, the maximum size of any batch cannot be greater than z. Since the response time cannot be greater than the length of the busy interval an upper-bound on response time given an upper-bound on L is as follows:

$$t_{busy}^{end} - t_{busy}^{start} \leq (\hat{L} + 1) \cdot b(z) \tag{9.2.6.7}$$

p_{min} **Upper-Bound.** Rather than brute-force search, a more direct upper bound calculation is possible by considering the minimum period of all tasks under consideration. Deriving a bound from using the previous Lemma 9.2.4 requires one to find a positive integer that is a bound on the number of batches released in any busy interval. Intuitively, the smallest integer possible is preferred, which can accomplished by incrementally testing integers starting with one. The problem is that we do not have a bound on the number of integers that must be tested. However, a direct upper bound calculation for L can be achieved by considering the minimum period, p_{min} of all tasks. Such an approach is useful provided that all tasks have the same or similar period. An example of such a case where all the periods are likely to be the same is a multimedia server, where each client stream has similar data request patterns.

Let $p_{min}(\mathrm{T})$ denote the smallest period of any task in T.

Lemma 9.2.5

if

$$p_{min}(\mathrm{T}) \geq b(z) + \left\lfloor \frac{\sum_{i=1}^{n} e_i}{z} \right\rfloor b(z)$$

then

$$L \leq \left\lfloor \frac{\sum_{i=1}^{n} e_i}{z} \right\rfloor + 1$$

Proof.

From Lemma 9.2.4, any positive integer x that does not satisfy the following inequality is a valid upper-bound on L:

$$\sum_{i=1}^{n} \left\lceil \frac{b(S_{c_{in}}) + (x-1) \cdot b(z)}{p_i} \right\rceil e_i > x \cdot z \tag{9.2.6.8}$$

Substituting $b(S_{c_{in}})$ by the upper bound $b(z)$ gives,

$$\sum_{i=1}^{n} \left\lceil \frac{x \cdot b(z)}{p_i} \right\rceil e_i > x \cdot z \tag{9.2.6.9}$$

Choose $x = \left\lfloor \frac{\sum_{i=1}^{n} e_i}{z} \right\rfloor + 1$ and substitute for x in the above inequality,

$$\sum_{i=1}^{n} \left\lceil \frac{(\lfloor \sum_{i=1}^{n} e_i \rfloor + 1) b(z)}{p_i} \right\rceil e_i > \left(\left\lfloor \sum_{i=1}^{n} e_i \right\rfloor + 1 \right) z \tag{9.2.6.10}$$

Substituting $p_{min}(\mathrm{T})$ for any p_i can only increase the left hand side of the inequality and therefore is a valid substitution giving

$$\sum_{i=1}^{n} \left\lceil \frac{\left(\left\lfloor \frac{\sum_{i=1}^{n} e_i}{z} \right\rfloor + 1 \right) \cdot b(z)}{p_{min}(\mathrm{T})} \right\rceil e_i > \left(\left\lfloor \sum_{i=1}^{n} e_i \right\rfloor + 1 \right) z \tag{9.2.6.11}$$

Substituting the right-hand side of the hypothesis, $p_{min}(\mathrm{T}) > b(z) + \lfloor \sum_{i=1}^{n} e_i \rfloor b(z)$, for $p_{min}(\mathrm{T})$,

$$\sum_{i=1}^{n} \left\lceil \frac{\left(\left\lfloor \frac{\sum_{i=1}^{n} e_i}{z} \right\rfloor + 1 \right) \cdot b(z)}{\lfloor \sum_{i=1}^{n} e_i \rfloor b(z) + b(z)} \right\rceil e_i > \left(\left\lfloor \sum_{i=1}^{n} e_i \right\rfloor + 1 \right) z \tag{9.2.6.12}$$

The top and bottom of the fraction inside the ceiling are equivalent and therefore reduce to one,

$$\sum_{i=1}^{n} e_i > \left(\left\lfloor \frac{\sum_{i=1}^{n} e_i}{z} \right\rfloor + 1 \right) z \tag{9.2.6.13}$$

Removing the floor through the inequality relationship, $\lfloor x \rfloor \geq x - 1$,

$$\sum_{i=1}^{n} e_i > \left(\frac{\sum_{i=1}^{n} e_i}{z} \right) z \tag{9.2.6.14}$$

Canceling common factors z in the numerator and denominator of the right-hand-side term gives

$$\sum_{i=1}^{n} e_i > \sum_{i=1}^{n} e_i \tag{9.2.6.15}$$

Therefore, a value of $x = \lfloor \sum_{i=1}^{n} e_i \rfloor + 1$ does not satisfy the inequality of Lemma 9.2.4 so,

$$L \leq \left\lfloor \frac{\sum_{i=1}^{n} e_i}{z} \right\rfloor + 1 \tag{9.2.6.16}$$

□

Lemma 9.2.6

if

$$p_{min}(\mathrm{T}) > b(z) + \left\lfloor \frac{\sum_{i=1}^{n} e_i}{z} \right\rfloor b(z)$$

then

the maximum number of requests released and served in an arbitrary busy interval can be no greater than $\sum_{i=1}^{n} e_i$

Proof.

All requests served in the busy interval are released in the time interval $[t_{busy}^{start}, \sigma_L]$, the length of which can be calculated as

$$\sigma_L - t_{busy}^{start} = b(S_{c_{in}}) + \sigma_L - \sigma_1 \tag{9.2.6.17}$$

Each batch dispatched in the busy interval excluding B_L contains z requests. Therefore, $L - 1$ batches contain z requests. The requests in these $L - 1$ batches are served in the time interval

(σ_1, σ_L). An upper-bound on the length of (σ_1, σ_L) is the maximum time to serve $L-1$ batches each containing z requests, which can be described by the following inequality

$$\sigma_L - \sigma_1 \leq (L-1)b(z) \tag{9.2.6.18}$$

Substituting $\sigma_L - \sigma_1$ by $(L-1)b(z)$ in Equation 9.2.6.17 gives

$$\sigma_L - t_{busy}^{start} \leq b(S_{c_{in}}) + (L-1)b(z) \tag{9.2.6.19}$$

From Lemma 9.2.5 and the hypothesis bound on p_{min}, $L \leq \left\lfloor \frac{\sum_{i=1}^{n} e_i}{z} \right\rfloor + 1$. Substituting for L gives

$$\sigma_L - t_{busy}^{start} \leq b(S_{c_{in}}) + \left\lfloor \frac{\sum_{i=1}^{n} e_i}{z} \right\rfloor b(z) \tag{9.2.6.20}$$

The maximum batch size is z, so $S_{c_{in}} \leq z$. Substituting $S_{c_{in}}$ by z gives,

$$\sigma_L - t_{busy}^{start} \leq b(z) + \left\lfloor \frac{\sum_{i=1}^{n} e_i}{z} \right\rfloor b(z) \tag{9.2.6.21}$$

The hypothesis lower bound on $p_{min}(\mathrm{T})$ is greater than the right-hand side of the above inequality. As such, the periodic constraint limits each task to releasing no more than one job in $[t_{busy}^{start}, \sigma_L]$. It follows then that the maximum number of requests released and served in an arbitrary busy interval can be no greater than $\sum_{i=1}^{n} e_i$. □

Let S_{back} denote the number of requests released at or prior to an arbitrary job J_k. That is, $S_{back} = Rel\Big([t_{busy}^{start}, R_k]\Big)$.

Theorem 9.2.7

if

$$p_{min}(\mathrm{T}) > b(z) + \left\lfloor \frac{\sum_{i=1}^{n} e_i}{z} \right\rfloor b(z)$$

then

$$t_{busy}^{end} - t_{busy}^{start} \leq \left(\left\lfloor \frac{\sum_{i=1}^{n} e_i}{z} \right\rfloor + 1 \right) b(z) + b\left(\sum_{i=1}^{n} e_i \bmod z \right)$$

Proof.

By the definition of a busy interval,

$$t_{busy}^{end} - t_{busy}^{start} \leq b(S_{c_{in}}) + \left\lfloor \frac{S_{back}}{z} \right\rfloor \cdot b(z) + b(B_L) \tag{9.2.6.22}$$

Since B_L cannot be greater than z,

$$t_{busy}^{end} - t_{busy}^{start} \leq b(S_{c_{in}}) + \left\lfloor \frac{S_{back}}{z} \right\rfloor \cdot b(z) + b(min(z, B_L)) \tag{9.2.6.23}$$

Let S_{rem} denote the remainder of the backlog served in B_L, so the final batch of a busy interval is the sum of freeloaders and the remainder of the backlog $B_L = S_{free} + S_{rem}$. Substituting for B_L,

$$t_{busy}^{end} - t_{busy}^{start} \leq b(S_{c_{in}}) + \left\lfloor \frac{S_{back}}{z} \right\rfloor \cdot b(z) + b(min(z, S_{free} + S_{rem})) \tag{9.2.6.24}$$

Given the lower bound on p_{min} in the hypothesis, Lemma 9.2.6 states that the maximum number of requests released and served in a busy interval is $\sum_{i=1}^{n} e_i$. Since the backlog and freeloaders are the only requests released and served in the busy interval,

$$S_{back} + S_{rem} + S_{free} \leq \sum_{i=1}^{n} e_i \tag{9.2.6.25}$$

Solving for $S_{rem} + S_{free}$,

$$S_{rem} + S_{free} \leq \sum_{i=1}^{n} e_i - S_{back} \tag{9.2.6.26}$$

Substituting $S_{rem} + S_{free}$ in Equation 9.2.6.24 by the right-hand-side of the above inequality gives

$$t_{busy}^{end} - t_{busy}^{start} \leq b(S_{c_{in}}) + \left\lfloor \frac{S_{back}}{z} \right\rfloor \cdot b(z) + b\left(min\left(z, \sum_{i=1}^{n} e_i - S_{back}\right)\right) \tag{9.2.6.27}$$

The maximum batch size is z, so substituting $S_{c_{in}}$ by z gives,

$$t_{busy}^{end} - t_{busy}^{start} \leq b(z) + \left\lfloor \frac{S_{back}}{z} \right\rfloor \cdot b(z) + b\left(min\left(z, \sum_{i=1}^{n} e_i - S_{back}\right)\right) \tag{9.2.6.28}$$

Factoring out $b(z)$ from $b(z) + \left\lfloor \frac{S_{back}}{z} \right\rfloor \cdot b(z)$ gives

$$t_{busy}^{end} - t_{busy}^{start} \leq \left(\left\lfloor \frac{S_{back}}{z} \right\rfloor + 1\right) b(z) + b\left(min\left(z, \sum_{i=1}^{n} e_i - S_{back}\right)\right) \tag{9.2.6.29}$$

The maximum over all valid S_{back} values is still a valid upper-bound on the length of the busy interval,

$$t_{busy}^{end} - t_{busy}^{start} \leq \max_{1 \leq S_{back} \leq \sum_{i=1}^{n} e_i} \left(\left(\left\lfloor \frac{S_{back}}{z} \right\rfloor + 1\right) b(z) + b\left(min\left(z, \sum_{i=1}^{n} e_i - S_{back}\right)\right)\right) \tag{9.2.6.30}$$

The following reasons that $S_{back} = \sum_{i=1}^{n} e_i - \sum_{i=1}^{n} e_i \bmod z$ is the value that maximizes the right-hand-side of the above inequality. From Lemma 9.2.6, the maximum number of requests released and served in a busy interval is $\sum_{i=1}^{n} e_i$. Therefore, S_{back} can be no greater than $\sum_{i=1}^{n} e_i$.

Consider the left and right terms containing S_{back}. Increasing S_{back} monotonically increases the resulting value of the left term. S_{back} values of $(\sum_{i=1}^{n} e_i - \sum_{i=1}^{n} e_i \bmod z)$ to $(\sum_{i=1}^{n} e_i)$ do not change the resulting left term value and result in the maximum value of the left term.

In contrast, decreasing S_{back} from $(\sum_{i=1}^{n} e_i)$ to $(\sum_{i=1}^{n} e_i - \sum_{i=1}^{n} e_i \bmod z)$ monotonically increases the resulting value of the right term, but the increase in the right term's resulting value is limited to $b(z)$ due to the *min*.

Any value of S_{back} smaller than $\sum_{i=1}^{n} e_i \bmod z$ can only result in a summed total of the left and right terms equal to or less than that the value at $S_{back} = \sum_{i=1}^{n} e_i$ to $\sum_{i=1}^{n} e_i - \sum_{i=1}^{n} e_i \bmod z$ since decreasing S_{back} to any value smaller than $\sum_{i=1}^{n} e_i \bmod z$ will decrease the left term's value by at least $b(z)$, but the increase in the right term's value is limited to $b(z)$.

Substituting $\sum_{i=1}^{n} e_i - \sum_{i=1}^{n} e_i \bmod z$ for S_{back} and removing the *max* based on the above reasoning gives,

$$t_{busy}^{end} - t_{busy}^{start} \leq \left(\left\lfloor \frac{\sum_{i=1}^{n} e_i - \sum_{i=1}^{n} e_i \bmod z}{z} \right\rfloor + 1 \right) b(z) + b\left(min\left(z, \sum_{i=1}^{n} e_i - (\sum_{i=1}^{n} e_i - \sum_{i=1}^{n} e_i \bmod z) \right) \right) \quad (9.2.6.31)$$

Combining common $\sum_{i=1}^{n} e_i$ terms

$$t_{busy}^{end} - t_{busy}^{start} \leq \left(\left\lfloor \frac{\sum_{i=1}^{n} e_i - \sum_{i=1}^{n} e_i \bmod z}{z} \right\rfloor + 1 \right) b(z) + b\left(min\left(z, \sum_{i=1}^{n} e_i \bmod z \right) \right) \quad (9.2.6.32)$$

Removing the *min* since $\sum_{i=1}^{n} e_i \bmod z$ cannot be larger than z,

$$t_{busy}^{end} - t_{busy}^{start} \leq \left(\left\lfloor \frac{\sum_{i=1}^{n} e_i - \sum_{i=1}^{n} e_i \bmod z}{z} \right\rfloor + 1 \right) b(z) + b\left(\sum_{i=1}^{n} e_i \bmod z \right) \quad (9.2.6.33)$$

Since $\frac{x - x \bmod z}{z} = \lfloor \frac{x}{z} \rfloor$ and $\lfloor \lfloor x \rfloor \rfloor = \lfloor x \rfloor$,

$$t_{busy}^{end} - t_{busy}^{start} \leq \left(\left\lfloor \frac{\sum_{i=1}^{n} e_i}{z} \right\rfloor + 1 \right) b(z) + b\left(\sum_{i=1}^{n} e_i \bmod z \right) \quad (9.2.6.34)$$

□

9.3 Evaluation of DQF on a HDD

The following section measures the performance of hard disk requests scheduled according to the DQF scheduling policy. The measured results are then compared with other scheduling algorithms and system configurations.

9.3.1 Amortized Service Time Model

The amortized service model used in computing the theoretical upper bounds on response time is described in Chapter 7. More precisely, the worst-case service times correspond to the maximum service times observed (labeled 1.0) in Figure 7.9.

9.3.2 Benchmarking Application

A synthetic user-space application was developed to measure experienced request service times. Once started, this application periodically asynchronously issues *count* randomly chosen read requests of a given size. The *period*, *read size*, and *count* are configurable parameters. A typical experiment starts some number of synthetic applications, each simultaneously competing for the hard disk to serve their respective request(s). It should be noted that *count* requests are issued at the start of every period regardless of whether or not previous requests have been served. The ability to perform multiple outstanding requests from the application is provided through the asynchronous I/O interface (rather than blocking I/O issued from threads).

9.3.3 Pseudo Block Device

A simulated hard disk device, implemented by a software device driver, was developed to increase the frequency of encountering worst-case behavior. In a GPOS, hard disk requests are most often submitted by user space applications. The submission time and location on disk of these requests tends to be highly variable. As such, experiencing worst-case service times also tends to be highly variable. There may be time intervals when service times are very short and any scheduling algorithm is sufficient for meeting deadlines. However, it is the time intervals when service times tend to be long that result in missed deadlines. While average case performance is considered in the design of DQF, the distinguishing feature of DQF is its ability to bound response times and thereby guarantee deadlines. Therefore, to more effectively demonstrate, compare, and examine the DQF's ability to bound response times, a software disk was developed to deterministically invoke worst-case behavior.

The software-based hard disk, we refer to as a *pseudo-disk*, imitates the characterized worst-case service timing behavior as discussed in Chapter 7. The pseudo-disk is an extension to the sample disk driver presented in the book *Linux Device Drivers* [22], which they call *sbull*. Figure 9.4 illustrates where the pseudo-disk resides in relation to an actual hard disk.

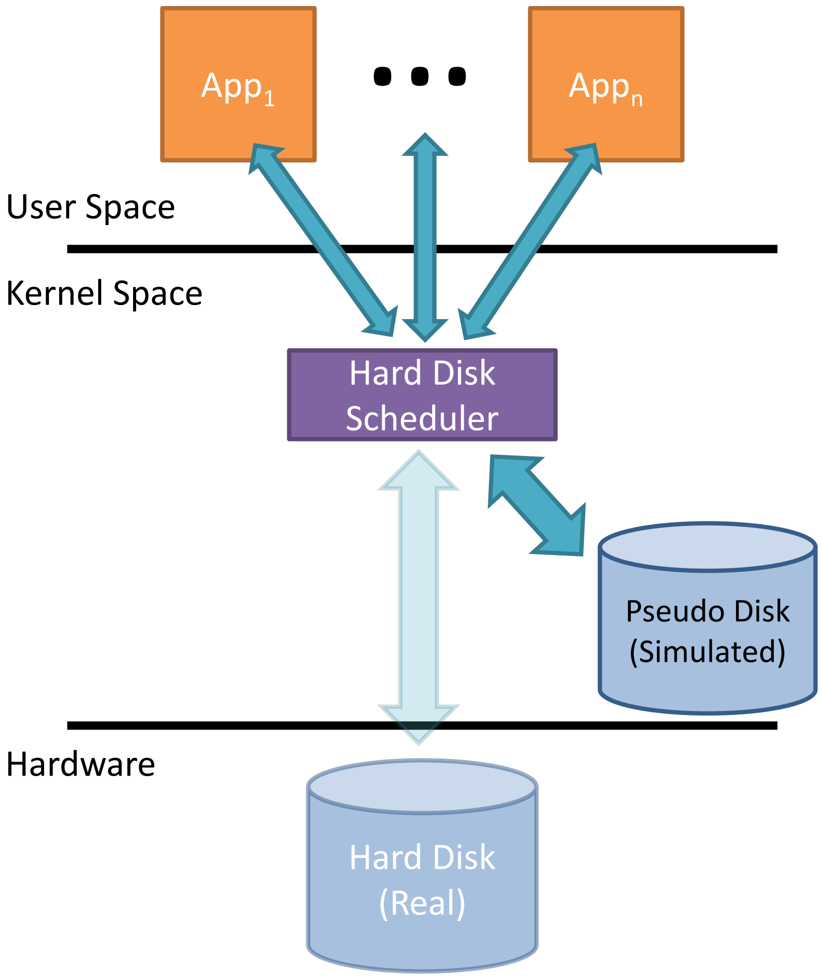


Figure 9.4: Diagram of the experimental system with the pseudo block device.

The pseudo-disk postpones a given request by calculating the response time using our hypothesized amortized worst-case Equation 7.4.1.7 with the corresponding values listed in Table 7.2. Upon receiving a batch of requests, the pseudo disk calculates the batch service time, considering the number of requests in the batch. The completion times of the requests are evenly spaced over the full batch service time. The pseudo disk sets a timer for each request completion time and at the timer's expiration it returns a randomly chosen request in the batch. The current timer is not altered if a new request is dispatched while the pseudo disk is serving 1 or more requests. The reasoning behind this is to match the typical behavior of an actual hard disk. That is, from our experience, once the disk picks the next request and starts positioning the disk head for it, the disk will completely service the chosen request, regardless of any subsequent requests that are dispatched. However, at the completion of the current request, the disk may decide on a new request service order. This behavior is duplicated in the pseudo-disk by recalculating the batch service time once the current timer expires. Note the recalculating of batch service times approximates request starvation and should not be experienced by a properly implemented DQF scheduler.

The pseudo disk returns arbitrary data on a read and discards any write data. This decision is partly due to the practical consideration that a sufficient amount of main memory is not available to record writes. However, even if write data is stored, the synthetic benchmarking applications do not use the returned data on a read nor expect the written data to be read back at a later time. Therefore, regardless of the data read/written, the synthetic applications should not experience any adverse effects.

9.3.4 Empirically Measured Response Times

This section presents experimental measurements to verify the credibility of our DQF upper bound analysis, worst-case service time estimates, and to compare DQF performance with other scheduling algorithms.

The initial workload chosen for evaluating DQF was a set of several tasks, each issuing a single 128KB read request every 300msec. The location of each request is randomly, uniformly distributed (across the entire disk). With a chosen workload, the upper bound on response times from Section 9.2 was computed.

Figure 9.5 plots the calculation using various values of z to provide some insight as to the effect of z on the response time. Also plotted is a trivial *lower bound* on the maximum response time, which is obtained by considering the case where a job of interest arrives immediately after all other tasks issue a request. In this case, the backlog (including the job of interest) is $\sum_{i=1}^{n} e_i$. Assuming service of such a backlog begins immediately upon its arrival, the maximum response time of the job of interest can be no less than the time to service all requests (additional requests may arrive and contribute to the maximum response time as freeloaders).

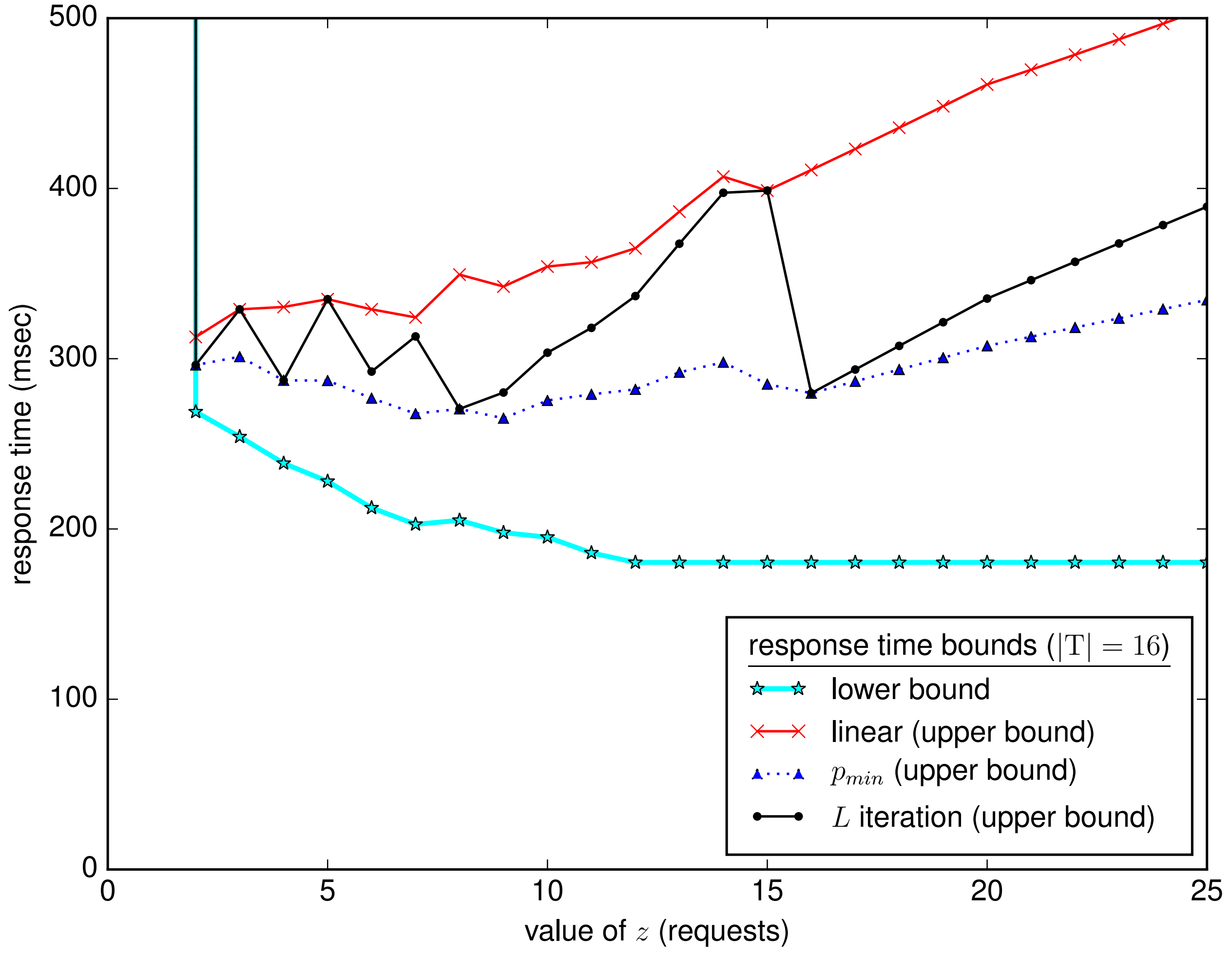


Figure 9.5: Comparison of derived upper and lower bounds using maximum observed response times. Each sporadic task τ_i has values $e_i = 1$ and $p_i = 300msec$.

The plot indicates that both the linear and refined upper bounds have certain values of z that are best suited for the chosen workload. Intuitively, the better z values illustrate the anticipated trade-off between latency and throughput. That is, as the batch size increases, larger throughput results along with lower response times. However, at some point, the increase in batch size results in increased non-preemption, thereby increasing the response time. The experiments that follow will generally use a maximum batch size of 8 (i.e., $z = 8$). This size provides provide a good lower bound guarantee for both the linear and refined upper bound calculations. Although the value of $z = 8$ is not the optimal choice for the linear bound, this value provides a reasonable guarantee for both bounds, thereby allowing us to validate them using the same experimental data.

We performed an initial evaluation of DQF by measuring the response times experienced by several instances of a synthetic application that simulates a periodic task with a 300msec period. A periodic rather than sporadic task was chosen for the following experiments with the intention invoking near worst-case scenarios by maximizing the system load. Two separate experiments were performed: one with the Linux noop (FIFO) scheduler and the other with our implemented DQF scheduler, where $z = 8$ (this z value was chosen based on earlier results). The reason for choosing the Linux noop is to provide a relatively effortless, but rough comparison with EDF. That is, noop will order service of requests the same as EDF when all tasks have the same relative deadlines. An upper bound on response time for EDF (noop) can be computed using the upper bound response time calculation for non-preemptive EDF given by George et al. [32]. Comparisons with the Linux CFQ I/O scheduler are presented later. Figure 9.6 shows the cumulative distribution of the measured response times for requests issued by 16 and 30 sporadic tasks, where each task submitted a total of 10,000 requests. The thin, vertical lines are used to highlight the maximum measured response time with the specified scheduler. The results provide initial evidence that scheduling via DQF maintains the guaranteed upper bound on response time. EDF performs better in terms of average-case response time than DQF where the number of tasks is 16. However, under the heavier load where the number of tasks is 30, DQF scheduling results in higher throughput than EDF and therefore, the relative average-case response time between EDF and DQF is not as pronounced. The improved average-case response time of DQF over EDF is the result of EDF not having sufficient throughput to serve submitted requests over extended periods of time. It should be noted that for this workload and EDF scheduling a theoretical upper bound does not exist using the worst-case service time of a single request since the utilization is greater than 1. However, the maximum measured response time for the observed measurements shown in Figure 9.6 is bounded since a worst-case sequence of service times corresponding to the worst-case of 1 request is not experienced. As such, the observed response times appear to be bounded, however, it is likely that the larger number of observations the larger the maximum observed measurement will be. That is, the response time is unbounded. This is in contrast to DQF in the case of T = 16, where the theoretical response time is bounded. Therefore, increasing the number of observed measurements will only increase the response time up to the theoretical limit, assuming the hard disk worst-case service time model is accurate.

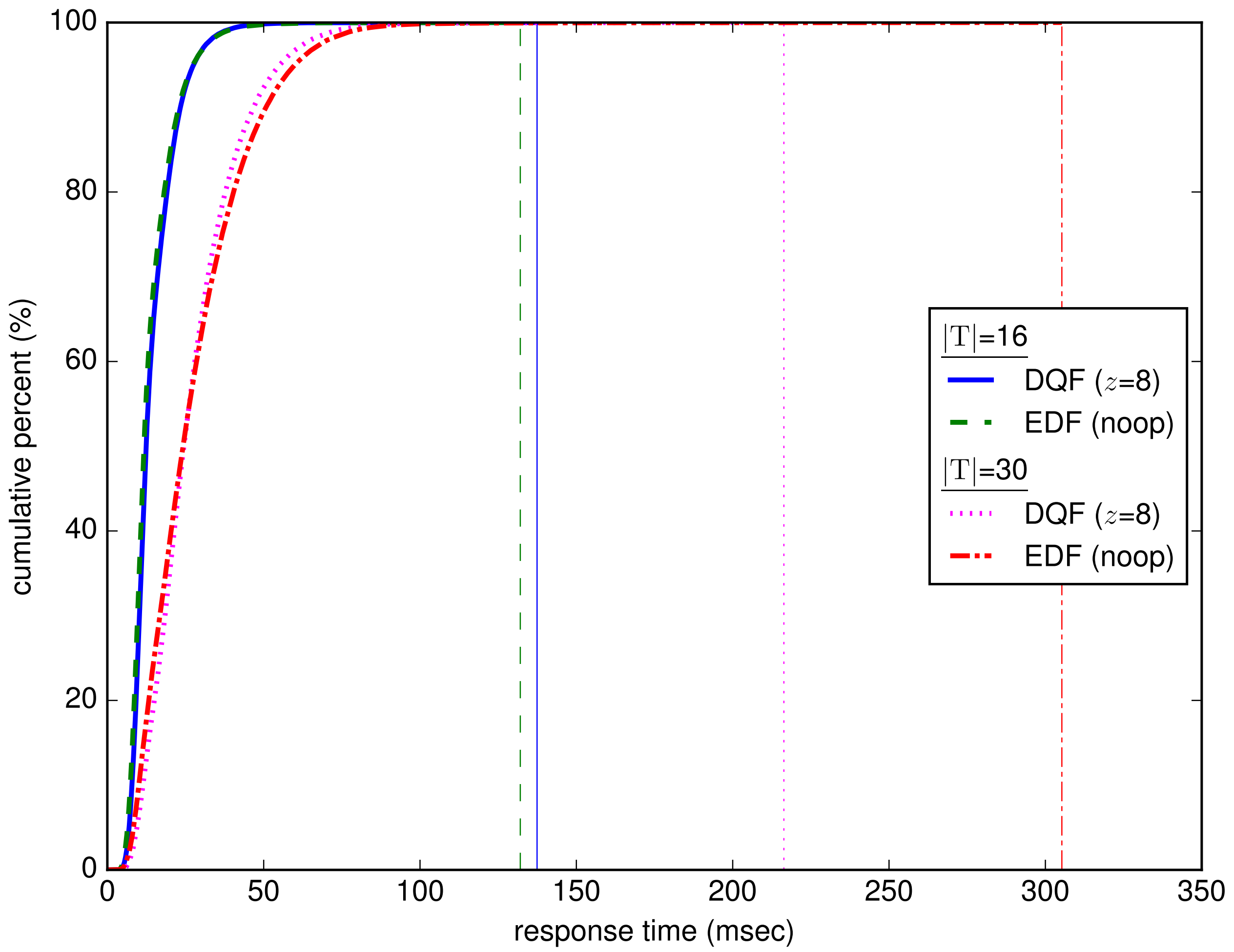


Figure 9.6: Response time distributions for DQF and FIFO with requests served on a hard disk.

The preceding experiment released all tasks at the same time instant and therefore it is almost certain that multiple requests will queue in the OS. One could reasonably conclude that such a situation favors the performance of DQF. Therefore, the experiment was altered slightly to change the pattern of tasks arriving at the same time by altering the period of each task. More explicitly, each task in the task set is assigned a unique value i from 0 to $n-1$ where n is the number of tasks in the task set. The period of each task is set to 300msec + i msecs. For example, for a task set of 30 tasks, one task is assigned a period of 300msecs, another 301msecs, another 302 msecs, and so on. The data for these varied period tasks is shown in Figure 9.7.

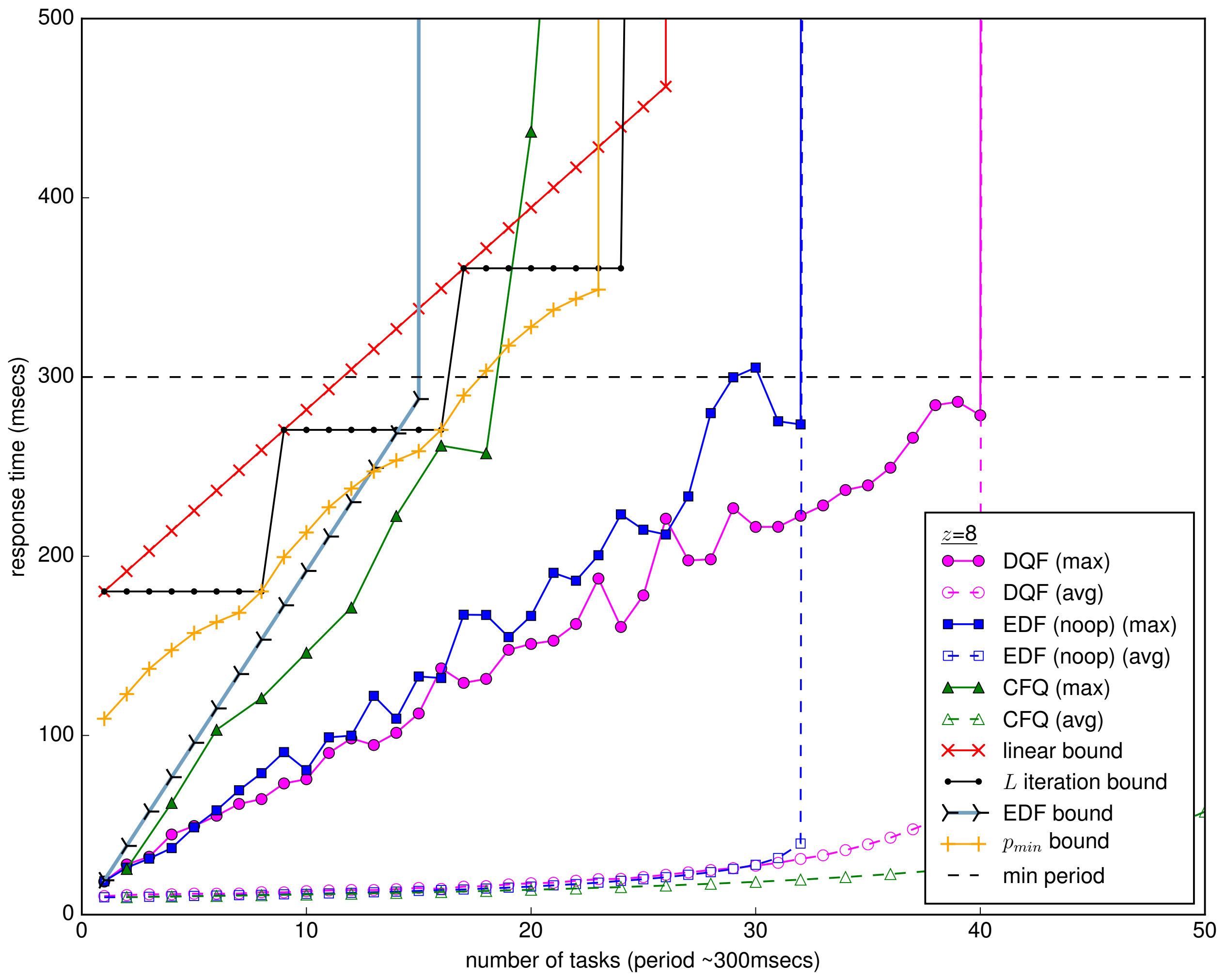


Figure 9.7: Comparison of DQF against other scheduling algorithms with requests served on a hard disk.

Increasing the number of tasks on the system will eventually result in the workload exceeding the disk's capacity for any reasonably long time interval. The term *overload* is used here to indicate such a condition. The vertical lines in Figure 9.7 are indicative of an overload condition. Note that once the system is overloaded, the response time increases dramatically and the largest measured response time largely depends on the length of time the experiment is run. The response time effect can be thought of as a positive feedback loop between increasing the request queue length and increasing waiting time of newly issued requests.

Experiments were run on the pseudo-disk to better illustrate the improvement of DQF. One of the distinguishing characteristics of DQF is lower, analytically bounded response times by taking advantage of increased throughput. In order to demonstrate this improvement, measured response time data using the pseudo-disk with the same experimental setup as the previous disk-based experiment with varied periods was performed and the results are plotted in Figure 9.8. It should be noted that the worst-case observed behavior described above is only a loose estimate of a hard disk's actual worst-case behavior. In contrast, the average measured service times experienced using the pseudo disk are very likely to be greater than an actual disk over long time intervals. However, the purpose of using the pseudo disk is to examine a given scheduler's impact on worst-case behavior, which may occur during run time.

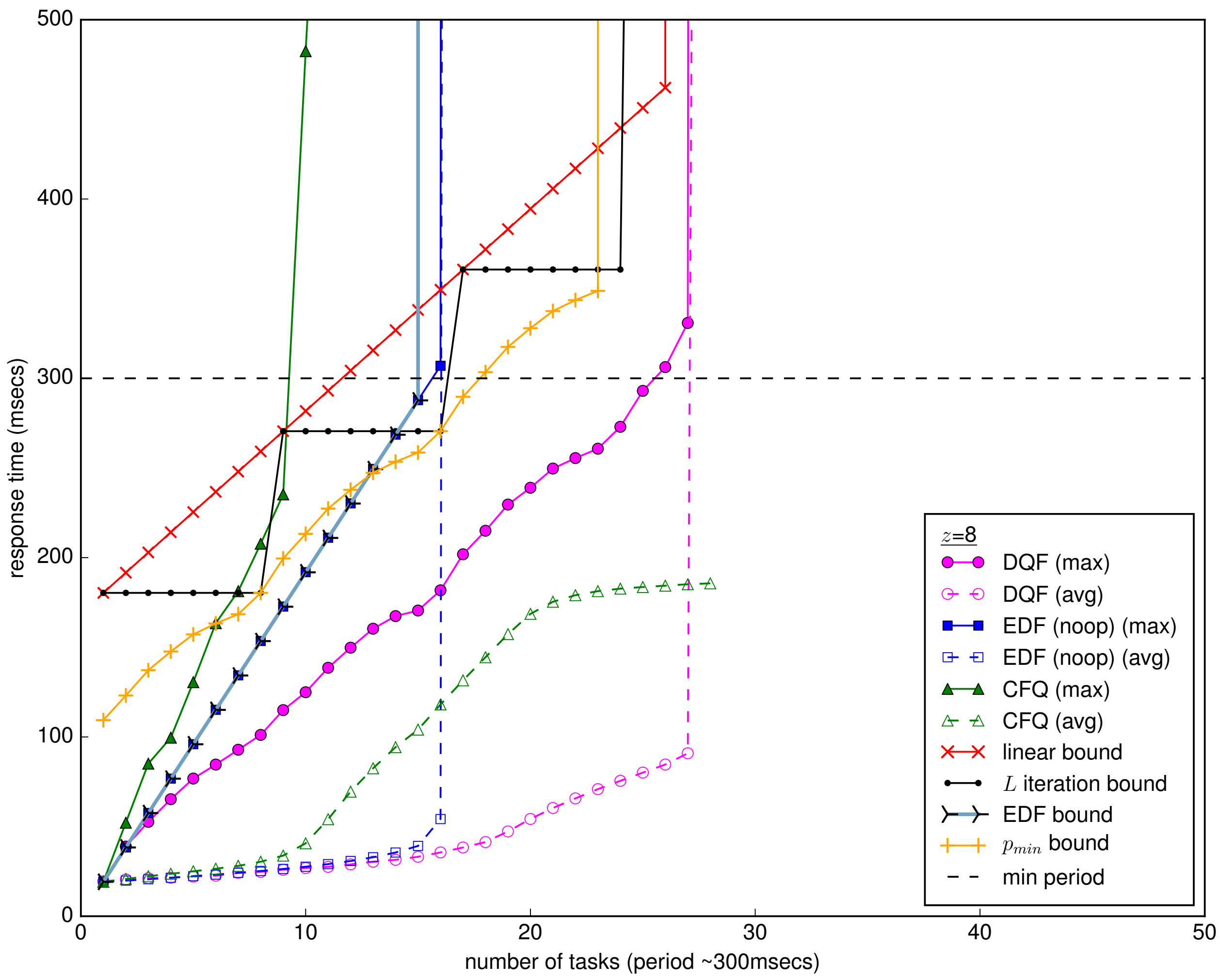


Figure 9.8: Comparison of DQF against other scheduling algorithms with requests served on a pseudo disk.

CFQ is the default Linux scheduler, which is largely composed of heuristics that provide generally good performance, but no known bound on response time. Figure 9.8 illustrates the potential to experience unpredictably large response times when using CFQ and why it is ill-suited if guarantees on maximum response times are desired. CFQ has many different tunable parameters and one can imagine that specific configurations would result in some bounded response times. However, to our knowledge, no analysis exists to configure CFQ for guaranteed response time bounds. Further, due to the complexity of CFQ, finding a tractable response time analysis seems unlikely. In contrast, we have derived analytical upper bounds on response time for DQF. Further, the simplicity of DQF increases the confidence that an implemented scheduling algorithm closely matches the theoretical assumptions.

The results of the Linux CFQ scheduler do illustrate a potential deficiency of DQF with respect to overload behavior. When an overload condition occurs, graceful degradation, meaning that the increase in response time be proportional to the extent of the overload, is generally desired. The maximum response times for requests served by DQF tend to rapidly increase with only minor overload conditions. A potential solution is to increase the permitted batch size dispatched to the disk, similar to the intuition behind the design of the polling and sporadic server hybrid described in Chapter 6. The idea would be to detect the overload condition and subsequently increase the maximum allowable batch size while the overload condition persists. The expected result is an increase in throughput that proportionally increases response time. Therefore, even though the guaranteed response time may be exceeded, the worst-case (and average-case) response time would not nearly as large as the current behavior as shown in Figures 9.7 and 9.8.

9.4 Related Work

9.4.1 Request Grouping

The SCAN-EDF [78, 92] algorithm is an extension to EDF that serves requests with the same deadlines in SCAN order. Our approach is similar to SCAN-EDF in that both leverage amortized seek time to improve time constrained performance. SCAN-EDF relies on request deadlines being the same and therefore requires all tasks to have a multiple of a predetermined base period. The effect is that SCAN-EDF picks the period, rather than the application choosing the period. In contrast our solution allows applications to choose any period (guarantees are still subject to meeting the schedulability analyses criteria) and further allows sporadic tasks rather than the restricted arrival pattern of a periodic task.

Li et al. proposed an algorithm for scheduling non-preemptive jobs on a uniprocessor that combines EDF and shortest job first (SJF) called Group-EDF (g-EDF) [62, 63]. The goal of g-EDF is to improve performance on soft real-time systems, primarily measured in the form of success rates of jobs that meet their deadline. Loosely speaking, the idea of g-EDF is to form groups of jobs whose deadlines are close together. Within each group, requests are ordered according to shortest job first. More specifically, the next job to execute is chosen as the one with the shortest execution time among those that have a deadline near to the earliest deadline of all available jobs. A job is considered near if its absolute deadline is no greater than the absolute deadline of the earliest deadline job plus the relative deadline of the earliest job multiplied by a predetermined factor. This predetermined factor is chosen based on experimentation.

While the authors of g-EDF demonstrate improved performance, there are at least two notable differences compared with DQF. First, the g-EDF work does not provide any analytical basis for *guaranteeing* time constrained performance and therefore, as noted by others [80], classifies g-EDF as a heuristic. The second differentiating characteristic of our work is that it is demonstrated on real hardware and in the context of a widely used general-purpose operating system. The demonstration of improved performance for g-EDF is provided only by the results of simulation experiments run using randomly generated jobs in MATLAB. The importance of experimenting with real hardware is that simulations often rely on models that are very loose approximations of the actual hardware. As such, especially without analytical guarantees, the expected behavior on an actual running system is very questionable.

Feasible Group EDF (Fg-EDF) [1] is a slightly altered version of g-EDF that is designed to better schedule hard disk I/O requests. The same algorithm as g-EDF is used to form groups. However, rather than shortest-job first, shortest-seek-time-first algorithm is used to order the requests in a group. An assumption appears to be made that seek time, as with shortest job, information is available, however, the authors do not comment on how such information is obtained. The "feasible" adjective of the algorithm describes a feasibility check performed before dispatching a given request for service. If the predicted service time of the request exceeds its deadline, the request is terminated without being served. The only demonstration of the performance of Fg-EDF is through a few hand-worked examples.

9.4.2 Leveraging Built-In Schedulers

One of the advantages of our technique is that it leverages a hard disk's built-in scheduler rather than attempting to perform similar scheduling solely in the OS. Many researchers have investigated scheduling real-time and best-effort hard disk requests in the OS. Some examples include Shenoy and Vin [104], Bosch, Mullender and Jansen [14], and Bosch and Mullender [15]. However, the majority of such studies do not consider leveraging the internal disk scheduler. Also, many – for example, see Reddy and Wyllie [92] and Cheng and Gillies [20] – require detailed knowledge of the disk's internal state, as well as its layout and timing characteristics experimentally. Such information is becoming increasingly difficult to obtain with the rapid growth in complexity and evolution of disk drives.

Others [72, 96, 97] have shown that it is possible to extract disk characteristics and perform fine-grained external scheduling. In fact, they were able to out-perform the on-disk scheduler in some cases. However, determining such timing information from disks can be very challenging and time-consuming, and can be expected to become more so as disk drives become more sophisticated. Moreover, this information is subject to change as the disk wears and sectors are relocated.

Fine-grained CPU-based disk scheduling algorithms require that the OS accurately track the disk's state. With a real-time system, such tracking would compete for CPU time along with other real-time application tasks [10, 122]. Therefore, it may be necessary to schedule the processing of disk I/O requests by the OS at a lower priority than some other tasks on the system. Interference by such tasks may prevent a CPU-based disk scheduling algorithm from accurately tracking the disk's state in real time, even if the layout and timing characteristics of the disk are known. Also, if the on-disk queue is not used, there is a risk of leaving the disk idle while it waits for the next request (i.e., the thread that issues requests to the disk), thereby diminishing the performance of the I/O service with a severity proportional to the amount of time that the issuing thread is delayed from receiving CPU time.

Internal disk scheduling algorithms do not have such difficulties with obtaining CPU time since they are executed by the disk's dedicated processor, and they have immediate access to the disk's internal state and timing characteristics. Even if the issuing thread is run at maximum priority, an off-disk scheduler will likely have (1) less complete and less timely information, (2) a reduced amount of control due to the limited information provided by the disk I/O interface, and (3) contention and transmission delays through the intervening layers of bus and controller.

The need to consider the internal scheduler of disks has been discussed in [50], which uses a round-based scheduler to issue requests to the disk. This scheduler allows real-time requests to be sent to the disk at the beginning of each round. The SCSI ordered tag is then used to force an ordering on the real-time requests. This approach prevents interference of requests sent after the real-time requests. However, using the ordered tag may impose unnecessary constraints on the on the built-in disk scheduler, which reduces the flexibility of the disk to make scheduling decisions and decreases throughput.

Another thread of research on real-time disk scheduling is represented by Wu and Brandt [120]. Noting the increasing intelligence of disk drives, they have taken a feedback approach to scheduling disk requests. When a real-time application misses its deadline, the rate of issuing the best-effort requests is reduced. While their work provides a way to dynamically manage the rate of missed deadlines, it does not provide precise a priori response time guarantees.

9.4.3 Techniques for Deriving Upper Bounds on Response Time

Others [53–55] have derived upper bounds for periodic task sets in the context of the economic lot scheduling problem. Periodic tasks tend to provide better guaranteed throughput since batching requests are can be deterministically predicted to arrive together. The obvious case is where all tasks start at the same time instant and have the same period. Therefore, at each period interval the number of newly released request is simply the number of tasks. On the other hand, with sporadic task sets, the release times of jobs may vary due to the period being only a minimum on the inter-release time. Therefore, jobs of a task may be released at slight offsets to one and other resulting in seeming smaller guaranteed groups of requests known to form a batch.

Network calculus [58] provides insights into properties of network flows and closely resembles arrival and service of hard disk requests considered in this chapter. In fact, network calculus has been applied to RT processor scheduling and is often referred to as real-time calculus [19]. While applying the techniques of network calculus to tighten our derived upper bound on response time seems plausible, we did not find a way to apply it in order to derive a tighter bound.

9.5 Conclusion

This chapter discusses a novel technique termed dual-queue that is used to modify an OS block-layer scheduling algorithm to leverage the throughput of hard disk's built-in scheduler while still meeting timing constraints. The effectiveness of dual-queue is demonstrated in this chapter by enhancing FIFO scheduling with dual-queue, resulting in a new scheduler termed dual-queue FIFO (DQF). Analysis is presented that derives a theoretical upper-bound on response time for DQF. Response time measurements are presented to illustrate the performance improvement achievable using DQF scheduling.

One of the advantages of the DQF scheduling algorithm is in its simplicity. Using on-line scheduling algorithms that are straight-forward to implement is a common trend for general-purpose operating systems. For example, earliest-deadline first (EDF) and fixed-task priority are two scheduling algorithms commonly encountered. As such, dual-queue is in-line with this trend since implementing a FIFO scheduling algorithm with dual-queue adds very little implementation complexity. Part of the reason for the trend of simple on-online scheduling algorithms is likely due to the difficulty of matching the implemented version of the scheduling algorithm with that of the theoretical algorithm. Any discrepancy between the two is likely to result in invalidating the theoretical guarantees. Therefore, one is generally more confident that a simple scheduling algorithm can be implemented to match its theoretical counterpart rather than a more complex one.

Improved performance may be obtainable by applying dual-queue to other real-time scheduling algorithms (e.g., EDF). The intuition is that EDF would send requests with tighter timing constraints earlier. However, dual-queue would still ensure that the performance of a hard disk's built-in scheduler is leveraged.

While implementing such a modified scheduler is certainly feasible, the primary difficulty is deriving a reasonable theoretical upper-bound on response time (or guarantee on meeting deadlines). The difficulty for us is providing a reasonable bound on the number of requests that violate the EDF scheduling order.

The presented system in this chapter assumes that the tasks issuing tasks are well-behaved. That is, each task does not issue violate its sporadic constraint used for analysis. Certainly in an actual system a misbehaving task could issue many requests and therefore jeopardize all other jobs' deadlines. A reasonable approach would be to use an aperiodic server to schedule requests on a per-task or per-group of tasks basis. In fact, such an approach would allow non-real-time applications that are not programmed with as a sporadic task to be incorporated into the system without change.

CHAPTER 10

FUTURE WORK

The experience and results achieved through the approach of balancing throughput and latency suggests future research including: investigating its effectiveness using other I/O devices 10.1, improving Dual-Queue Fifo analyses 10.2 statistical tools to provide a broader notion of soft-real-time guarantees 10.3, and improving performance as well as providing guarantees when the implemented system deviates from the theoretical model 10.4.

10.1 Other I/O Devices

Broadly speaking, two classes of I/O are addressed in this dissertation: 1) those whose maximum achievable real-time performance is largely dependent on CPU time scheduling (network) and 2) those whose maximum achievable real-time performance is largely dependent on scheduling peripheral device requests (hard disks). However, other peripheral I/O devices such as solid state disks (SSD) are likely to require both proper CPU and device scheduling in order to maximize its real-time performance capabilities. Considering hard disks, the ratio of CPU time to I/O device time per request is very small. Therefore, allocating such small slivers of CPU time is easily accomplished without affecting other tasks using the CPU. However, SSDs are able to serve requests more quickly in the worst-case than hard disks and therefore, the ratio of CPU to I/O time is significantly larger than that of hard disks. However, unlike the network peripheral device, the ordering of requests to an SSD is likely to allow for either low latency or high throughput. Therefore, the real-time performance of such devices must consider account both CPU and device scheduling. That is, the realized performance is a function of both CPU and device scheduling.

10.2 Dual-Queue FIFO (DQF)

10.2.1 Tighter Derived Upper-Bound on Response Time

A tighter theoretical upper bound on response time than that derived in Section 9.2 is likely achievable. Figure 10.1 shows the performance of DQF for larger values of z than presented previously. In contrast to the prediction of our current upper bound on response time, larger values of z do not increase the maximum measured response time for the same workload, but in fact result in a lower response time. If it is possible to use larger values of z without increasing (or even better decreasing) the response time upper bounds, even broader timing guarantees could be supported (e.g., more tasks). However, the obstacle is deriving a schedulability analysis that can prove larger values of z do in fact reduce the maximum response time achievable (i.e., tighter worst-case response time bounds).

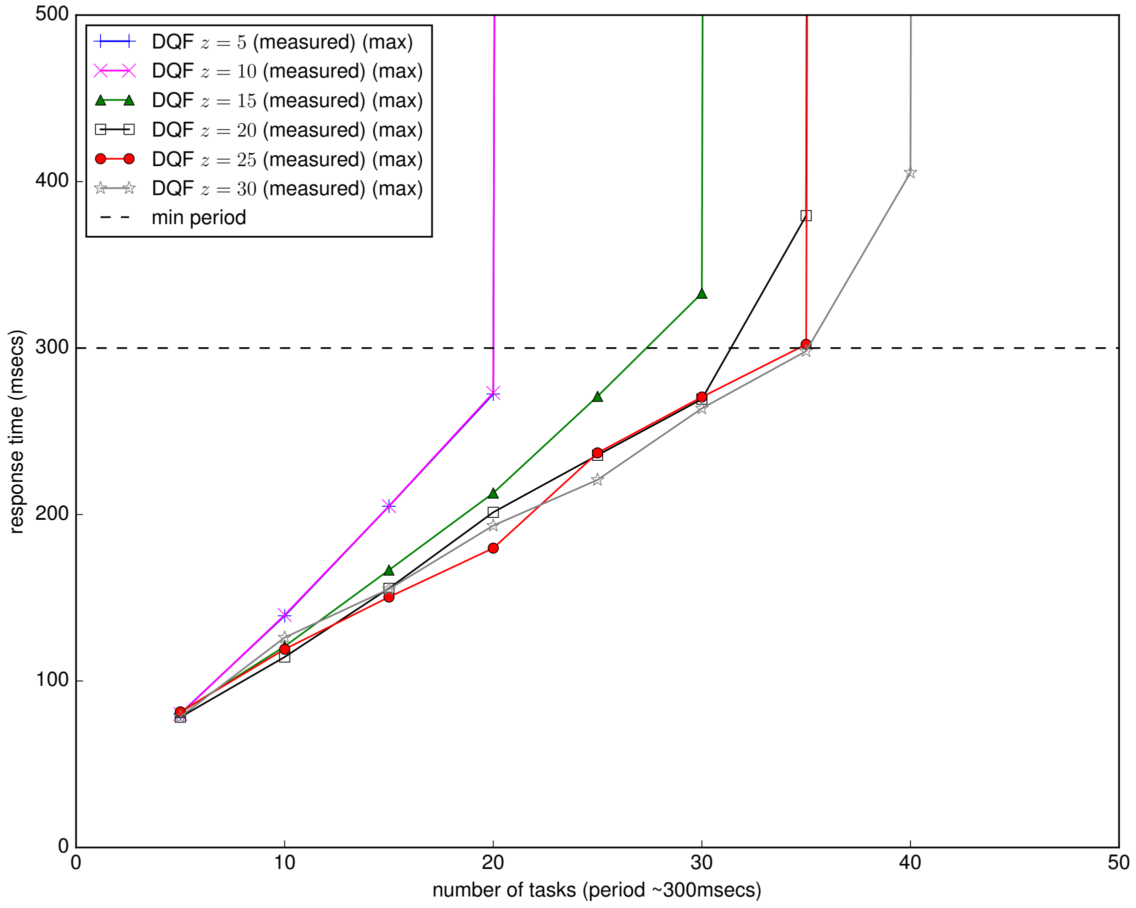


Figure 10.1: (pseudo disk) Maximum observed service times with larger values of z to illustrating likely conservatism with DQF derived upper bound.

A tighter upper bound for tasks with the same period may be achievable by slightly altering the technique of George et al. for deriving an upper bound on response time with non-preemptive EDF [32]. Since job priorities' assigned by FIFO and EDF scheduling are the same when tasks have the same period, the analysis for DQF should be nearly the same, with the exception of the amount of priority inversion. The technique of George et al. establishes a so-called “deadline busy period” for the task being analyzed to determine an upper bound on response time. The amount of priority inversion in the deadline busy period is limited to a single job. However, in the case of DQF, the priority inversion would not be limited to a single job but must also include the priority inversion of the freeloaders.

A further adaptation of [32] may also be possible even in the case where all tasks do not have the same period. In such an adaptation, a similar notion of the deadline busy interval would be defined to limit the interval required to search. Further, the time instants to consider would be the release time instants of other tasks that coincide, similar to their approach where deadlines coincide. The technique of George et al. has pseudo-polynomial time complexity.

10.2.2 DQF and Other Types of I/O

Dual-queue scheduling is likely to be effective on other I/O service besides hard disk I/O. One of the first devices that comes to mind is sold-state drives (SSD). SSDs tend to have much better performance characteristics, especially with respect to random workloads. The main hurdle to applying DQF (or likely any other scheduling algorithm) is to extract worst-case service profiles. Similar to hard disks, manufacturers generally do not disclose built-in scheduling algorithms. Commodity SSDs appear to be even more difficult than hard disks due to the built-in Flash Translation Layers that map logical to physical locations. However, unlike disks, the mappings change much more frequently. More specifically, hard disks will generally only change the logical address of a data location when the physical location degrades such that it is no longer able to reliably store data. However, with SSDs, the logical to physical mapping changes with nearly every write. Therefore, measured worst-case service time of a given request at the same logical location may be substantially different than a previous measurement. More concretely, the data of a given request may become spread out internally on the SSD, even though the logical mapping indicates that the data is stored contiguously.

CPU processing of I/O, as discussed in Section 6.6, may be able to take advantage of DQF scheduling analyses. For instance, imagine scheduling CPU time for receive packet processing with a polling server at the highest priority of the system. In this case, packets are served in varying sized batches depending on the frequency of packet arrivals. As discussed earlier, these batches have amortized worst-case service times. While DQF would not be used to directly schedule the processor time, it can be used to reason about an upper bound on response times for handing network packets to their destination process.

10.3 Probabilistic Analysis

Micro-level analysis of modern commodity hardware and software is generally unrealistic due to a large number of unknowns arising from the diversity and complexity of the system components. However, as discussed in this dissertation useful abstract models can be derived by observing empirical timing measurements under certain conditions, essentially providing macro-level analysis using the instantiated system itself.

The applications targeted for this research are expected to tolerate the “occasional” missed deadline. Part of the reason for the “soft” nature of the real-time system is that the validity of analytical guarantees hinge on approximate worst-case service times. There are unknown amounts of uncertainty in the observed measurements and even the listed data sheet values. By applying statistical and probabilistic techniques the worst-case service times can be adjusted with more confidence. On the one hand, one could more effectively reason about the amount to increase the worst-case service times to account for uncertainties. For instance, the branch of statistics known as extreme value theory has been applied to the probability of encountering a particular worst-case execution time value. On the other hand, one could also reason about reducing the worst-case service times used in the analysis by defining better metrics to quantify and more effectively define the “softness” of the guarantee.

10.4 Graceful Degradation

One can easily imagine many discrepancies in the assumptions of the theoretical model with the real-world. Such discrepancies may appear for any number of reasons including: aging effects (e.g., bad sectors), environmental factors (e.g., electromagnetic interference), and unanticipated system interactions, just to name a few. However, when such unexpected factors occur it is often desirable that the system continue operation and that the effects on timing (e.g., length of time by which a deadline is missed) is known and proportional to the effect due to the encountered unknown during run time such that the system is able to continue operation rather than failing completely. Often many systems can continue to provide reasonable functionality. The discussions on the advantages of the scheduling techniques (e.g., dual-queue) presented in this dissertation have alluded to such a property, but have not been rigorously studied. Future research includes how the trade-off between throughput and latency can be leveraged to gracefully degrade rather than fail completely.

CHAPTER 11

CONCLUSIONS

This chapter concludes the dissertation by summarizing the contributions and discussing possible extensions to the presented line of research that still remain to be explored.

11.1 Summary of Contributions

The research of this dissertation set out to adapt a general-purpose operating system's (GPOS) I/O scheduling to balance throughput and latency with the intention of broadening real-time support. The general focus of most research on real-time design tends to focus solely on low latency without considering the negative impact on throughput. On the other hand, many general purpose designs focus on high throughput and average-case performance ignoring the impact on timely performance. This dissertation demonstrates that a GPOS can be practically adapted to allow a better balancing of throughput and latency in order to meet and guarantee I/O timing constraints.

The initial research of this dissertation set out to simply fit the CPU component of I/O service into an abstract model usable by existing real-time analyses. I/O service in a GPOS relies on the CPU and therefore contends with other real-time applications. Timing guarantees provided by real-time scheduling theory results generally rely on describing processor usage in terms of abstract workload models. Therefore, without appropriately formulated workload models the applying existing schedulability analyses is not possible. As such, this dissertation research started by developing a measurement-based technique to formulate an appropriate abstract workload model for CPU usage of I/O activities, thereby allowing timing guarantees to be established for both I/O services other real-time applications contending usage of the system's CPU.

I/O services provided by GPOS are generally not designed to meet timing constraints and therefore occasionally consume large amounts of CPU time in small time intervals. Since, schedulability analyses consider worst-case behavior these periods of large CPU time consumption tend to unnecessarily prevent many other applications from meeting their deadlines. Therefore, sporadic server, a fixed-task priority aperiodic server scheduling algorithm, was implemented in order allow one to better control the CPU time consumption of I/O services with respect to other applications contending for CPU time.

The validity of the schedulability analyses results depend on an accurate match between the implementation and the theoretical assumptions. Our implementation of sporadic server initially followed the specification defined in well-known Single UNIX Specification (SUS), which resulted in the discovery of several errors in the specification. The intention of sporadic server is to conform demand for CPU time in such a way to allow one use a simple periodic task workload model for analysis. However, we discovered that implementing sporadic server as specified in the SUS results in CPU demand that does not provide the intended equivalence to a periodic task. The SUS version of sporadic server is a translation of the theoretical version of sporadic server with modifications to account for practical considerations (e.g., overruns). These modifications resulted in a deviation from the theoretical model and thereby invalidating the ability to model the CPU demand as an equivalent periodic task. We identified these errors in the SUS sporadic server and provided corrections to achieve the equivalence to a periodic task.

CPU time consumed by processor intensive I/O services, such as network I/O, is often too focused on latency and ignores the adverse impact on CPU processing throughput. Sporadic server allows arbitrarily small CPU time slices to be used with the intention of lowering latency, however, for I/O service this often comes at the cost of reduced throughput. Some amount of time elapses when switching between threads on a physical a CPU. While the time can certainly be included in the analysis, this switching time does not result in activities making progress. In particular, I/O processing tends to use small slices of CPU time resulting in large amounts of CPU time spent switching rather than processing I/O. Under light load, this switching reduces latency, however, under heavy load, this switching results in substantial reductions in CPU time allocated to process I/O, thereby decreasing I/O throughput.

A novel hybrid aperiodic server is introduced as part of this dissertation research to allow sporadic server to dynamically adjust between low latency and high throughput. The intuition is that servicing processor time spent switching to service I/O is acceptable and desired when the I/O presented workload is light since it tends to decrease the average-case response time. However, switching very often during heavy load reduces the amount of I/O that is processed since processor time is wasted switching between other activities and I/O service(s). The novel hybrid server acts the same as a sporadic server under light I/O load. However, the load increases to a point that the sporadic server uses all its budget, the slices of time allocated to sporadic server are coalesced. Under sufficiently high I/O loads, the coalesced time (budget) essentially degrades into a polling server. Therefore, the server is a hybrid between the polling and sporadic aperiodic server scheduling algorithms.

The trade-off between latency and throughput is extended to peripheral I/O devices through a novel scheduling algorithm termed Dual-Queue FIFO (DQF). In addition to requiring the CPU, providing I/O service also requires peripheral hardware devices. A hard disk is used to demonstrate the trade-off between latency and throughput. The service of batches of (multiple) I/O requests by the hard disk incurs a similar penalty as with processor time slicing. Improved throughput is achievable when larger batches are permitted. With the hard disk, this additional minimum guaranteed throughput is the result of amortizing the seek time over multiple hard disk requests. That is, issuing multiple requests (batches) of requests to the hard disk results in a deterministically lower worst-case service time than if requests were scheduled individually. The Dual-Queue FIFO scheduling algorithm allows one to configure a system to adjust the degree to which a commodity devices' throughout-based scheduling is leveraged through a single batch size parameter. Associated theoretical analysis is derived to appropriately determine an upper bound on response time. This upper-bound allows one to appropriately configure the DQF batching parameter with respect to the applications' desired timing constraints.

TRADEMARKS

UNIX is a registered trademark of the Open Group. LINUX is a registered trademark of Linus Torvalds.

BIBLIOGRAPHY

[1] S. Y. Amdani and M. S. Ali. “An Improved Group-EDF: A Real-Time Disk Scheduling Algorithm,” *International Journal of Computer Theory and Engineering*, 5(6):873–876, Dec. 2013.

[2] Dave Anderson, Jim Dykes, and Erik Riedel. “More Than an Interface–SCSI vs. ATA,” in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, pp. 245–257, USENIX Association, Mar.–Apr. 2003.

[3] M. Andrews, M. A. Bender, and L. Zhang. “New Algorithms for Disk Scheduling,” *Algorithmica*, 32(2):277–301, Feb. 2002.

[4] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. “Bounding Worst-Case Instruction Cache Performance,” in *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pp. 172–181, Dec. 1994.

[5] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces.* 0.80. Arpaci-Dusseau Books, May 2014.

[6] N. C. Audsley, A. Burns, M. Richardson, and A. J. Wellings. “Hard Real-Time Scheduling: The Deadline Monotonic Approach,” in *Proc. of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, pp. 127–132, 1991.

[7] J. Axboe. *CFQ (Complete Fairness Queueing).* `http://www.kernel.org`. 2015 [accessed 01-Jan-2015].

[8] J. Axboe. “Linux Block IOâĂŤpresent and future,” in *Proceedings of the Linux Symposium (OLS ’04)*, pp. 51–61, July 2004.

[9] Theodore P. Baker and Sanjoy K. Baruah. “Schedulability Analysis of Multiprocessor Sporadic Task Systems,” in *Handbook of Real-time and Embedded Systems.* Ed. by Insup Lee, Joseph Y-T. Leung, and Sang Son. CRC Press, 2007.

[10] Theodore P. Baker, An-I Andy Wang, and Mark J. Stanovich. “Fitting Linux Device Drivers Into an Analyzable Scheduling Framework,” in *Proc. of the 3rd Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pp. 1–9, July 2007.

[11] T.P. Baker. “An Analysis of EDF Schedulability on a Multiprocessor,” *IEEE Transactions on Parallel and Distributed Systems*, 16(8):760–768, Aug. 2005.

[12] Sanjoy K. Baruah, A. K. Mok, and L. E. Rosier. "Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor," in *Proc. of the 11th IEEE Real-Time Systems Symposium (RTSS '90)*, pp. 182–190, Dec. 1990.

[13] G. Bernat and A. Burns. "New Results on Fixed Priority Aperiodic Servers," in *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS '99)*, pp. 68–78, Dec. 1999.

[14] P. Bosch, S.J. Mullender, and P.G. Jansen. "Clockwise: A Mixed-Media File System," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS '99)*, pp. 277–281, IEEE Computer Society, July 1999.

[15] Peter Bosch and Sape J. Mullender. "Real-Time Disk Scheduling in a Mixed-Media File System," in *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS '00)*, pp. 23–32, IEEE Computer Society, May–June 2000.

[16] Reinder J. Bril and Pieter J. L. Cuijpers. *Analysis of Hierarchical Fixed-Priority Pre-Emptive Scheduling Revisited.* Tech. rep. CSR-06-36. Eindhoven, Netherlands: Technical University of Eindhoven, 2006.

[17] A. Burns, A.J. Wellings, and A.D. Hutcheon. "The impact of an Ada run-time system's performance characteristics on scheduling models," in *Ada - Europe '93*. Ed. by Michel Gauthier. Vol. 688. *(Lecture Notes in Computer Science)*. Springer Berlin Heidelberg, 1993, pp. 240–248.

[18] Alan Burns. "Preemptive priority-based scheduling: an appropriate engineering approach," in *Advances in real-time systems*, pp. 225–248, Prentice-Hall, Inc., 1995.

[19] Samarjit Chakraborty, Simon Kunzli, and Lothar Thiele. "A general framework for analysing system properties in platform-based embedded system designs," in *In Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE '03)*. Vol. 1, pp. 190–195, Mar. 2003.

[20] Raymond M. K. Cheng and Donald W. Gillies. "Disk management for a hard real-time file system," in *Proc. of the Eighth Euromicro Workshop on Real-Time Systems*, pp. 255–260, June 1996.

[21] Jonathan Corbet. *SCHED_FIFO and realtime throttling.* `http://lwn.net/Articles/296419/`. Sept. 2008.

[22] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition.* O'Reilly Media, Inc., 2005.

[23] R. I. Davis and A. Burns. "Hierarchical Fixed Priority Preemptive Scheduling," in *Proc. 26th IEEE Real-Time Systems Symposium (RTSS '05)*, pp. 376–385, Dec. 2005.

[24] R.I. Davis and A. Burns. "Resource Sharing in Hierarchical Fixed Priority Pre-Emptive Systems," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS '06)*, pp. 257–270, Dec. 2006.

[25] Z. Deng and J. W. S. Liu. "Scheduling real-time applications in an open environment," in *Proc. of the 18th IEEE Real-Time Systems Symposium*, pp. 308–319, Dec. 1997.

[26] Sarah Diesburg, Christopher Meyers, Mark Stanovich, Michael Mitchell, Justin Marshall, Julia Gould, An-I Andy Wang, and Geoff Kuenning. "TrueErase: Per-File Secure Deletion for the Storage Data Path," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, pp. 439–448, Dec. 2012.

[27] Jake Edge. *ELC: SpaceX lessons learned.* `http://lwn.net/Articles/540368/`. Mar. 2014.

[28] Jake Edge. *Moving interrupts to threads.* `http://lwn.net/Articles/302043/`. Oct. 2008.

[29] Dario Faggioli. *POSIX SCHED_SPORADIC implementation for tasks and groups.* `http://lkml.org/lkml/2008/8/11/161`. Aug. 2008.

[30] Dario Faggioli, Marko Bertogna, and Fabio Checconi. "Sporadic Server Revisited," in *SAC '10: Proceedings of 25th ACM Symposium On Applied Computing*, ACM, Mar. 2010.

[31] Greg Ganger. *DIXTRAC: Automated Disk Drive Characterization.* `http://pdl.cmu.edu/Dixtrac/index.shtml`. 2008.

[32] Laurent George, Nicolas Rivierre, and Marco Spuri. *Preemptive and Non-Preemptive Real-Time Uniprocessor Scheduling.* Research Report RR-2966. Projet REFLECS. INRIA, 1996.

[33] T. M. Ghazalie and Theodore P. Baker. "Aperiodic servers in a deadline scheduling environment," *Real-Time Systems*, 9(1):31–67, 1995.

[34] Thomas Gleixner. *Add support for threaded interrupt handlers - V3.* `http://lwn.net/Articles/324980/`. Mar. 2009.

[35] Google. *Android, the world's most popular mobile platform.* `http://developer.android.com/about/index.html`. 2014 [accessed 21-July-2014].

[36] Google. *Google TV.* `http://www.google.com/tv/`. 2014 [accessed 21-July-2014].

[37] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* 2nd. Addison-Wesley Longman Publishing Co., Inc., 1994.

[38] The Open Group. *Standards.* `http://www.opengroup.org/standards`. 2014 [accessed 01-Aug-2014].

[39] Nan Guan and Wang Yi. "General and efficient Response Time Analysis for EDF scheduling," in *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE '14)*, 255:1–255:6, Mar. 2014.

[40] Jeffery Hansen, Scott A. Hissam, and Gabriel A. Moreno. "Statistical-Based WCET Estimation and Validation," in *Proceedings of the 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[41] C.A. Healy, R.D. Arnold, F. Mueller, D.B. Whalley, and M.G. Harmon. "Bounding pipeline and instruction cache performance," *IEEE Transactions on Computers*, 48(1):53–70, 1999.

[42] IBM. *146GB 15K 3.5" Hot-Swap SAS.* `http://www-132.ibm.com/webapp/wcs/stores/servlet/ProductDisplay?catalogId=-840&storeId=1&langId=-1&dualCurrId=73&categoryId=4611686018425093834&productId=4611686018425132252`. [Online; accessed 04-October-2007]. 2007.

[43] IEEE Portable Application Standards Committee (PASC). *Standard for Information Technology - Portable Operating System Interface (POSIX) Base Specifations, Issue 7.* IEEE, Dec. 2008, pp. 1–3826.

[44] Intel. *82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC).* `http://download.intel.com/design/chipsets/datashts/29056601.pdf`. 1996.

[45] Intel. *Intel 64 Architecture (x2APIC) Specification.* `http://www.intel.com/Assets/pdf/manual/318148.pdf`. Mar. 2010.

[46] Michael B. Jones and Stefan Saroiu. "Predictability requirements of a soft modem," *SIGMETRICS Perform. Eval. Rev.* 29(1):37–49, 2001.

[47] Asim Kadav and Michael M. Swift. "Understanding Modern Device Drivers," in *Proc. of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*, pp. 87–98, 2012.

[48] Scott Karlin and Larry Peterson. *Maximum Packet Rates for Full-Duplex Ethernet.* Tech Report TR-645-02. Princeton University, Feb. 2002.

[49] D. I. Katcher, H. Arakawa, and J. K. Strosnider. "Engineering and analysis of fixed priority schedulers," *IEEE Transactions on Software Engineering*, 19(9):920–934, Sept. 1993.

[50] Kyung Ho Kim, Joo Young Hwang, Seung Ho Lim, Joon Woo Cho, and Kyu Ho Park. "A Real-Time Disk Scheduler for Multimedia Integrated Server Considering the Disk Internal Scheduler," in *Proc. of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03)*, pp. 124–130, IEEE Computer Society, Apr. 2003.

[51] Richard P. King. “Disk Arm Movement in Anticipation of Future Requests,” *ACM Transactions on Computer Systems (TOCS)*, 8(3):1–6, Aug. 1990.

[52] S. Kleiman and J. Eykholt. “Interrupts as Threads,” *ACM SIGOPS Operating Systems Review*, 29(2):21–26, Apr. 1995.

[53] J. Korst. “Periodic Multiprocessor Scheduling,” PhD thesis. Eindhoven: Eindhoven University of Technology, 1992.

[54] J. Korst, V. Pronk, E. Aarts, and F. Lamerikx. “Periodic scheduling in a multimedia server,” in *Proc. of the INRIA/IEEE Symposium on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1, pp. 205–216, Oct. 1995.

[55] Jan H. M. Korst, Emile H. L. Aarts, and Jan Karel Lenstra. “Scheduling Periodic Tasks with Slack.,” *INFORMS Journal on Computing*, 9(4):351–362, 1997.

[56] Elie Krevat, Joseph Tucek, and Gregory R. Ganger. “Disks are like snowflakes: no two are alike,” in *Proceedings of the 13th USENIX conference on Hot topics in operating systems (HotOS’13)*, USENIX Association, May 2011.

[57] Butler W. Lampson and David D. Redell. “Experience with processes and monitors in Mesa,” *Commun. ACM*, 23(2):105–117, 1980.

[58] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag, 2001.

[59] J. P. Lehoczky. “Fixed Priority Scheduling of Periodic Task Sets With Arbitrary Deadlines,” in *Proc. of the 11th IEEE Real-Time Systems Symposium (RTSS ’90)*, pp. 201–209, Dec. 1990.

[60] Rick Lehrbaum. *An Interview with Preemptible Kernel Patch Maintainer Robert Love*. `http://www.linuxjournal.com/article/5755`. [accessed August-2014]. 2002.

[61] Mark Lewandowski, Mark J. Stanovich, Theodore P. Baker, Kartik Gopalan, and An-I Wang. “Modeling Device Driver Effects in Real-Time Schedulability Analysis: Study of a Network Driver,” in *Proc. Of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 57–68, Apr. 2007.

[62] Wenming Li. “Group-Edf: A New Approach and an Efficient Non-Preemptive Algorithm for Soft Real-Time Systems,” PhD thesis. Denton, TX, USA: University of North Texas, Aug. 2006.

[63] Wenming Li, Krishna Kavi, and Robert Akl. “A non-preemptive scheduling algorithm for soft real-time systems,” *Computers and Electrical Engineering*, 33(1):12–29, 2007.

[64] Fujitsu Limited. *MAX3147RC MAX3073RC MAX3036RC Hard Disk Drives Product/Maintenance Manual.* `http://193.128.183.41/home/v3__ftrack.asp?mtr=/support/disk/manuals/c141-e237-01en.pdf`. 2005.

[65] Linux. *RT PREEMPT.* `https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO`. [accessed August-2013]. 2013.

[66] G. Lipari and E. Bini. "Resource partitioning among real-time applications," in *Proc. 15th EuroMicro Conf. on Real-Time Systems*, pp. 151–158, July 2003.

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

[68] J. W. S. Liu. *Real-Time Systems.* Prentice-Hall, 2000.

[69] C. Douglass Locke. "Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives," *Real-Time Syst.* 4(1):37–53, 1992.

[70] Rapita Systems Ltd. *Hybrid (measurement and static analysis).* `http://www.rapitasystems.com/WCET-Hybrid-Static-and-Measurement`. July 2012.

[71] Rapita Systems Ltd. *RapiTime On Target Timing Analysis.* `http://www.rapitasystems.com/rapitime`. May 2010.

[72] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. "Freeblock Scheduling Outside of Disk Firmware," in *Proceedings of the Conf. on File and Storage Technologies (FAST)*, USENIX, Jan. 2002.

[73] Maxtor. *Atlas 10K V.* `http://www.seagate.com/support/internal-hard-drives/enterprise-hard-drives/atlas-10k/`. [Online; accessed 08-January-2014].

[74] Paul McKenney. *A realtime preemption overview.* `http://lwn.net/Articles/146861/`. Aug. 2005.

[75] Larry McVoy and Carl Staelin. *LMbench - Tools for Performance Analysis.* `http://www.bitmover.com/lmbench/`. May 2010.

[76] Larry McVoy and Carl Staelin. "lmbench: Portable Tools for Performance Analysis," in *USENIX Annual Technical Conference*, pp. 279–294, Jan. 1996.

[77] J. Mogul and K. Ramakrishnan. "Eliminating Receive Livelock in an Interrupt-Driven Kernel," *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[78] A.L. Narasimha Reddy and J.C. Wyllie. "I/O issues in a multimedia system," *Computer*, 27(3):69–74, Mar. 1994.

[79] Kannan Narasimhan and Kelvin D. Nilsen. "Portable Execution Time Analysis for RISC Processors," in *in Proceedings of the Workshop on Architectures for Real-Time Applications*, 1994.

[80] Mitra Nasri, Sanjoy Baruah, Gerhard Fohler, and Mehdi Kargahi. "On the Optimality of RM and EDF for Non-Preemptive Real-Time Harmonic Tasks," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems (RTNS '14)*, pp. 331–340, Oct. 2014.

[81] Netflix. *Open Connect Appliance Hardware.* `https://www.netflix.com/openconnect/hardware`. 2014 [accessed 21-July-2014].

[82] Kelvin D. Nilsen and Bernt Rygg. "Worst-case execution time analysis on modern processors," in *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, & Tools for Real-Time Systems (LCTES)*, pp. 20–30, ACM, 1995.

[83] J. K. Ousterhout. "Why Aren't Operating Systems Getting Faster As Fast as Hardware?," in *Proceedings of the Usenix Summer 1990 Technical Conference*, pp. 247–256, June 1990.

[84] Yen-Jen Oyang. "A Tight Upper Bound of the Lumped Disk Seek Time for the Scan Disk Scheduling Policy," *Information Processing Letters*, 54(6):355–358, 1995.

[85] R. Pellizzoni and M. Caccamo. "Toward the Predictable Integration of Real-Time COTS Based Systems," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS)*, pp. 73–82, Dec. 2007.

[86] Anna Povzner, Tim Kaldewey, Scott Brandt, Richard Golding, Theodore M. Wong, and Carlos Maltzahn. "Efficient Guaranteed Disk Request Scheduling with Fahrrad," in *Proc. of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (Eurosys '08)*, pp. 13–25, ACM, Apr. 2008.

[87] LTTng Project. *Linux Trace Toolkit - next generation.* `http://lttng.org/`.

[88] Harilaos N. Psaraftis, Marius M. Solomon, Thomas L. Magnanti, and Tai-Up Kim. "Routing and Scheduling on a Shoreline with Release Times," *Management Science*, 36(2):212–223, Feb. 1990.

[89] Jin Qian, Christopher Meyers, and An-I Andy Wang. "A Linux implementation validation of track-aligned extents and track-aligned RAIDs," in *Proc. of the 2008 USENIX Annual Technical Conference (USENIX ATC '08)*, pp. 261–266, June 2008.

[90] Jin Qian and An-I Andy Wang. *A Behind-the-Scene Story on Applying Cross-Layer Coordination to Disks and RAIDs.* Tech. rep. TR-071015. Tallahassee, FL, USA: Florida State University Department of Computer Science, Oct. 2007.

[91] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems.* McGraw-Hill, Inc., 2003.

[92] A. L. Narasimha Reddy and Jim Wyllie. "Disk scheduling in a Multimedia I/O System," in *Proc. of the first ACM Int. Conf. on Multimedia (MULTIMEDIA '93)*, pp. 225–233, ACM Press, Aug. 1993.

[93] John Regehr. "Inferring Scheduling Behavior with Hourglass," in *Proc. of the USENIX Annual Technical Conf. FREENIX Track*, pp. 143–156, June 2002.

[94] John Regehr and Usit Duongsaa. "Preventing Interrupt Overload," in *Proc. 2006 ACM SIGPLAN/SIGBED Conf. on languages, compilers, and tools for embedded systems*, pp. 50–58, June 2005.

[95] John Regehr and John A. Stankovic. "Augmented CPU Reservations: Towards Predictable Execution on General-Purpose Operating Systems," in *Proc. of the 7th Real-Time Technology and Applications Symposium (RTAS 2001)*, pp. 141–148, May 2001.

[96] Lars Reuther and Martin Pohlack. "Rotational-Position-Aware Real-Time Disk Scheduling Using a Dynamic Active Subset (DAS)," in *Proc. of the 24th IEEE Int. Real-Time Systems Symposium (RTSS '03)*, pp. 374–385, Dec. 2003.

[97] C. Ruemmler and J. Wilkes. "An Introduction to Disk Drive Modeling," *Computer*, 27(3):17–28, Mar. 1994.

[98] Saowanee Saewong, Ragunathan (Raj) Rajkumar, John P. Lehoczky, and Mark H. Klein. "Analysis of Hierarchical Fixed-Priority Scheduling," in *Proceedings of the 14th Euromicro Conf. on Real-Time Systems (ECRTS)*, pp. 152–160, June 2002.

[99] Jiri Schindler and Gregory R. Ganger. "Automated Disk Drive Characterization (Poster Session)," in *Proc. of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '00)*, pp. 112–113, ACM, June 2000.

[100] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. "Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics," in *Proc. of the 1st USENIX Conference on File and Storage Technologies (FAST '02)*, USENIX Association, Jan. 2002.

[101] Sebastian Schönberg. "Impact of PCI-Bus Load on Applications in a PC Architecture," in *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS '03)*, pp. 430–439, IEEE Computer Society, Dec. 2003.

[102] L. Sha, R. Rajkumar, and J. P. Lehoczky. "Priority inheritance protocols: an approach to real-time synchronisation," *IEEE Trans. Computers*, 39(9):1175–1185, 1990.

[103] L. Sha, T. Abdelzaher, K. E. Årzén, A. Cervin, Theodore P. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. "Real Time Scheduling Theory: A Historical Perspective," *Real-Time Systems*, 28(2–3):101–155, Nov. 2004.

[104] Prashant J. Shenoy and Harrick M. Vin. "Cello: a disk scheduling framework for next generation operating systems," in *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint Int. Conf. on Measurement and modeling of computer systems*, pp. 44–55, ACM Press, 1998.

[105] Insik Shin and Insup Lee. "Compositional real-time scheduling framework with periodic model," *ACM Trans. Embed. Comput. Syst.* 7(3):1–39, 2008.

[106] B. Sprunt, L. Sha, and L. Lehoczky. "Aperiodic task scheduling for hard real-time systems," *Real-Time Systems*, 1(1):27–60, 1989.

[107] Mark J. Stanovich, Theodore P. Baker, and An-I Andy Wang. "Throttling On-Disk Schedulers to Meet Soft-Real-Time Requirements," in *Proc. of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '08)*, pp. 331–341, Apr. 2008.

[108] Mark J. Stanovich, Theodore P. Baker, and Andy An-I Wang. "Experience with Sporadic Server Scheduling in Linux: Theory vs. Practice," in *Proc. of the 13th Real-Time Linux Workshop (RTLWS)*, pp. 219–230, Oct. 2011.

[109] Mark J. Stanovich, Theodore P. Baker, Andy An-I Wang, and M. González Harbour. "Defects of the POSIX Sporadic Server and How to Correct Them," in *Proc. of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 35–45, Apr. 2010.

[110] M.J. Stanovich, I. Leonard, K. Sanjeev, M. Steurer, T.P. Roth, S. Jackson, and M. Bruce. "Development of a Smart-Grid Cyber-Physical Systems Testbed," in *Proc. of the 4th IEEE Conference on Innovative Smart Grid Technologies (ISGT)*, Feb. 2013.

[111] M.J. Stanovich, S.K. Srivastava, D.A. Cartes, and T.L. Bevis. "Multi-Agent Testbed for Emerging Power Systems," in *Proc. of the IEEE Power and Energy Society General Meeting (PES-GM)*, July 2013.

[112] D. B. Stewart. "More Pitfalls for Real-Time Software Developers," *Embedded Systems Programming*, Nov. 1999.

[113] D. B. Stewart. "Twenty-Five Most Common Mistakes with Real-Time Software Development," in *Embedded Systems Conference*, Sept. 1999.

[114] Daniel Stodolsky, J. Bradley Chen, and Brian N. Bershad. "Fast Interrupt Priority Management in Operating System Kernels," in *USENIX Microkernels and Other Kernel Architectures Symposium*, pp. 105–110, USENIX Association, Sept. 1993.

[115] T10 Technical Committee on SCSI Storage Interfaces. *SCSI Architecture Model - 3 (SAM-3)*. `http://www.t10.org/ftp/t10/drafts/sam3/sam3r14.pdf`. 2004.

[116] Vasily Tarasov, Gyumin Sim, Anna Povzner, and Erez Zadok. "Efficient I/O Scheduling with Accurately Estimated Disk Drive Latencies," in *Proc. of the 8th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2012)*, pp. 36–45, July 2012.

[117] Y. C. Wang and K. J. Lin. "The implementation of hierarchical schedulers in the RED-Linux scheduling framework," in *Proc. 12th EuroMicro Conf. on Real-Time Systems*, pp. 231–238, June 2000.

[118] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.* 7(3):1–53, 2008.

[119] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. "On-line Extraction of SCSI Disk Drive Parameters," in *Proc. of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95/PERFORMANCE '95)*, pp. 146–156, ACM, May 1995.

[120] Joel C. Wu and Scott A. Brandt. "Storage Access Support for Soft Real-Time Applications," in *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, pp. 164–171, IEEE Computer Society, 2004.

[121] J. Zhang, R. Lumia, J. Wood, and G. Starr. "Achieving Deterministic, Hard Real-Time Control on an IBM-compatible PC: A General Configuration Guideline," in *Proceedings of the 2005 IEEE Conference on Systems, Man and Cybernetics.* Vol. 1, pp. 293–299, Oct. 2005.

[122] Y. Zhang and R. West. "Process-Aware Interrupt Scheduling and Accounting," in *Proc. of the 27th IEEE Real Time Systems Symposium (RTSS)*, pp. 191–201, Dec. 2006.

[123] Peter Zijlstra. *sched: rt time limit.* `http://lkml.org/lkml/2007/12/30/258`. Dec. 2007.

BIOGRAPHICAL SKETCH

Mark J. Stanovich was born on April 26, 1978 in Wilkes-Barre, PA. Following high school, Mark enlisted into the U.S. Navy. After serving four years, he came to Florida State University as a college freshman, where he earned his BS and MS in computer science.