Automatic Monitoring for Interactive Performance and Power Reduction

by

Krisztián Flautner

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2001

To my grandmother, Eleonóra Majorosy.

Acknowledgements

I would like to thank my mother, Eleonóra Arató who, at first sceptical of me spending too much time with a brand new Commodore 64, eventually gave up trying to push me towards medical school and has been a steadfast supporter of my studies ever since. My wife, Krisztina Gerhardt helped me gather the focus to finish this dissertation. I thank my stepfather, Cecil Eby, whose adventures with computers have shown me time again that even simple triumphs over the machine can bring great satisfaction.

The research that eventually led to this dissertation began under Rich Uhlig's watch while I was at the Intel Microprocessor Research Lab in Portland, Oregon. His comments and ideas greatly influenced the initial direction. Steve Reinhardt has been very helpful at getting to the bottom of the issues and forcing me to elaborate on the details.

Conversations with and the compiler course taught by Peter Bird had a great impact on my view of computers. I would like to thank him for many afternoons' worth of interesting discussions. I would also like to thank my fellow graduate students, Matt Postiff and David Greene, who have been willing and helpful test subjects for many of the ideas in this dissertation.

Special thanks are owed to my advisor, Trevor Mudge, for supporting me and allowing me to explore a wide variety of subjects without constraining me to narrow objectives. While freedom can be daunting sometimes, I learned a lot during this process.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

This dissertation chronicles our investigations into two seemingly unrelated areas: thread-level parallelism and power management. However, these subjects are not as disconnected as they might seem at a first glance. In our work on thread-level parallelism we take a look at how effectively multiprocessing can be used to improve the response-times of interactive desktop applications, while our power management algorithm reduces power consumption when response-times are already fast enough. The key to both of these studies is a mechanism that can automatically find execution episodes with a direct impact on the user experience. We call these interactive episodes.

1.1 Overview

The impetus for developing a technique to distinguish interactive episodes from other parts of the execution came as a result of our initial experiments with measuring the thread-level parallelism of desktop applications [12]. While these initial investigations indicated that even in existing desktop applications, threads do indeed run concurrently, we were unable to answer one very important question: does the user perceive any improvement as a result of multiprocessing? In order to answer this question, we first needed to figure out exactly how to measure the user-perceived improvement.

Common benchmarking techniques that claim to assess the behavior of interactive applications often miss the point. For example the Winstone [49] and Sysmark [46] benchmarks both focus on how desktop applications behave with real-world usage scenarios, yet instead of directly measuring the impact on the user, they both measure the total execution times of their workloads. They accomplish this by using a software driver that simulates a human. However, instead of simulating a user who progresses at a “human” pace, the driver clicks and types through the application as quickly as possible. The length of the simulated interaction is measured and used as an indicator for performance. In essence, this technique takes an interactive workload and turns it into a throughput-oriented benchmark (e.g. SPEC2000 [45]).

The problem with this approach is that in interactive applications not all parts of the program's execution are equally important. The relevant metric for measuring the quality of performance is not end-to-end throughput, but response time: the time it takes the computer to respond to user initiated events [11]. This time has also been called "wait time" referring to the fact that during these episodes the user is actively waiting for the computer to complete the task.

The beginning of an interactive episode is initiated by the user and is usually signified by a GUI event, such as pressing a mouse button or a key on the keyboard. Finding the end of an episode is more difficult since there is no event that automatically gets generated when the computer is done responding. One approach is to assume that user initiated events are CPU bound and to define the end of an episode as the beginning of a relatively long idle section. The length of an interactive episode is thus the elapsed time between a user initiated event (e.g., a mouse click) and the beginning of the next idle period that is longer than a predefined threshold. There are two problems with this approach:

- Episodes that are I/O bound may be terminated prematurely if the wait time exceeds the idle threshold.
- There is a significant latency between the end of an interactive episode and its classification which complicates on-line use of episode information (e.g., for scheduling or power management).

We developed a more robust episode-detection mechanism to alleviate these problems. To find interactive episodes, we keep track of the set of tasks that communicate with each other as a result of a user-initiated GUI event (Chapter 4).

The start of an interactive episode is signified by the GUI controller process sending a message to another task. In return, this task performs work and may send messages to others and receive their answers. Once messages stop flowing between the tasks, the episode is over. Aside from fixing the problems associated with the approach the uses idle time to delineate the end of an episode, our episode-detection mechanism requires only a few kernel modifications and has a negligible run-time and memory overhead. We have also verified that interactive episodes are found accurately by comparing the timestamps of X server events with the begin and end timestamps of interactive episodes.

1.2 Thread-level parallelism

As processor architects design chips that are capable of concurrently running multiple threads [47][19], and the cost of multiprocessor systems come into the reach of mainstream users [2], it is important to consider what role multiprocessing can play on the desktop. Multiprocessing is already prevalent in servers where multiple clients present an obvious source of parallelism. However, the case for multiprocessing is not as clear for desktop users. It is not clear whether there is a sufficient number of concurrently runnable threads on the desktop, and whether the user experiences tangible benefits by taking advantage of them to justify the added cost of multiprocessing. Our goal was to find the answers to the following two questions:

- How much do threads run concurrently in interactive desktop applications?
- Does concurrency translate into improved interactive performance on a multiprocessor machine?

The initial investigations focus on a broad set of applications across three operating systems: BeOS, Linux, and Windows NT (Chapter 3). We developed a metric and a technique for measuring thread-level parallelism (TLP). Initially we started with using machine utilization as our metric but found that the large amount of idle time during interactive benchmarks can obscure the presence of concurrent execution. TLP is defined as the machine utilization over the non-idle portions of the benchmark's execution. This definition sidesteps the problems of the original metric and allows us to use more realistic workloads. Intuitively, TLP reflects the speedup due to concurrent execution on the non-idle portions of the workload. We found that while interactive applications tend to use a lot of threads, there are only moderate amounts of concurrency. While the measured TLP would not justify a quad-processor machine, we believe that dual-processor machines do have a place on the desktop.

We use our episode-detection technique to focus our investigations on the effects of multiprocessing on interactive episodes of Linux applications (Chapter 5). Our results show that the duration of perceptible interactive episodes is reduced by an average of 22%

as a result of using two processors instead of one. Moreover, the response time improvement increases to 29% if we include a concurrently running background process (an MP3 player) during our runs. We also observed that the majority of interactive episodes are very short (less than one millisecond) and that there is a significant amount of idle time (often more than 80%) during the runs of the benchmarks.

1.3 Automatic performance scaling

A key property of interactive episodes is that faster is not always better since there is a fundamental limit to what humans can perceive. Literature about human-computer interaction [37][5] indicates that 20-30 frames per second are sufficient for the human visual system to perceive the images as a continuous stream. This suggests that the perception threshold is around 50ms. Human subject tests in [5] show that perceptual causality—when two events are perceived to be fused together—ends around 100ms and for some test subjects quality degradation begins at around 50ms. Other experiments have shown that, for simple operations, such as dragging an object through the screen, as few as 5 updates per second are sufficient to maintain an interactive feel (200ms perception threshold). For non-continuous operations, as much as a 1-2 second delay is acceptable [37]. However, when human motor operations form a feedback loop with visual activity, it is more important to have a faster response time.

The fact that there is a limit to what humans can perceive forms the basis of our power management scheme. Instead of running interactive episodes as fast as possible, we aim to find an optimum performance level that stretches the duration of short interactive episodes exactly to the perception threshold. The algorithm only slows down episodes that are expected to be shorter than the perception threshold, long episodes are run at full performance to avoid further delays.

Power management algorithms in current mainstream operating systems do not vary the performance level of the processor: they either run programs at full performance or put the processor in idle mode. While these schemes avoid wasting energy when nothing is executing on the processor, they do not try to optimize energy use when the processor is utilized. Equation 1 shows a simplified equation relating power use to performance on a CMOS processor, where C is the capacitance, v is the operating voltage level, and f is the operating frequency.

$$P = Cv^2f \tag{EQ 1}$$

Moreover, the values for the voltage level and frequency are interrelated; the voltage level is proportional to the frequency of the processor. The key to dynamic voltage scaling is that, due to the quadratic voltage factor, even a small reduction in frequency can yield a significant reduction in power consumption. For example, the Intel 80200 [23] (based on the XScale microarchitecture, previously referred to as SA-2) processor requires 1.5V to run at 773Mhz but only 0.75V to run at 150Mhz. This implies that by running a workload at 150Mhz, one could use less than 1/20th of the power than at full performance. The impact of voltage scaling on energy savings is not as great as on power savings, since running at a lower performance level implies a proportional increase of total execution time. The energy used for executing a unit of work is proportional to the square of the operating voltage, which means that for a given workload running at 150Mhz requires a quarter of the energy of that running at 773Mhz.

The aim of our performance-scaling technique is to take advantage of power-saving features on processors such as the Intel XScale [23] and Transmeta Crusoe [25] that allow the frequency of the processor to be reduced with proportional reduction in voltage. Slowing down frequency without voltage scaling is not sufficient since the power savings is offset by an equal increase in execution time, yielding no reduction in the total amount of energy consumed [40].

The same episode-detection techniques, developed for measuring the effects of multiprocessing on response time, can be used to ensure that power management does not interfere with the quality of interactive performance. Moreover, we use episode-length information combined with a predictor as the basis for an automatic mechanism to estimate the optimum performance-level for program execution (Chapter 6). Unlike previous automated approaches, our method works equally well with irregular and multiprogrammed workloads. Our experiments show that, as a result of our algorithm, processor energy savings of as much as 75% can be achieved with only a minimal impact on the user experience.

1.4 Organization

Chapter 2 provides an overview of the previous work in our research areas. Chapter 3 gives the detailed results of our initial investigations into the nature and amount of thread-level parallelism of desktop applications. These results prompted the development of our episode-detection technique which is described in Chapter 4. We apply episode-detection to quantify the impact of multiprocessing on interactive performance in Chapter 5. The same methodology forms the basis of our automatic performance-setting algorithm for dynamic voltage scaling described in Chapter 6.

Chapter 2
Background

This chapter describes previous work in the areas addressed by the dissertation. The two main areas of background work are workload characterization and power management. We aim to provide a broad overview of these fields with an emphasis on papers that are directly relevant to our studies.

2.1 Characterization of interactive applications

Various papers have dealt with the characterization of desktop applications [27][6][11]. In [11], Endo et al. performed a detailed analysis of interactive performance in a uniprocessor Windows NT environment. In their paper they point out the need to measure response-time, instead of throughput for the entire benchmark, and explore the changes in response-times on different operating system versions and usage models. The authors classify the time during execution into two categories: think time and wait time. Think time is the time during which the user is neither making requests of the system nor waiting for the system to respond. Wait time is the time when the user is actively waiting for the system to do something, which is equivalent to the duration of interactive episodes in our studies.

Hauser et al. [20] have approached the role of threads in interactive systems by analyzing the design patterns in two threaded object-oriented environments: Cedar and GVX. Their analysis focuses on the use of threads for program structuring instead of run-time statistics. The authors identify ten different paradigms for using threads and conclude that most threads are used for programmers' convenience and few for exploiting concurrency. This observation is echoed by our results.

In [27], Lee et al. collect a wide variety of run-time statistics about the behavior of desktop applications. This study focuses on the impact of desktop applications on the processors by measuring TLB miss rate, branch prediction accuracy, cache effectiveness, etc. The authors observed that most of the instructions are executed from a single dominant thread. In our experience TLP can vary greatly based on the choice of OS and workloads. Our study of thread-level parallelism of desktop applications shows that multiprocessing can have beneficial effects even on standard desktop workloads.

2.2 Power management

While the literature on power management exposes a lot of ideas about what to do when the system is idle, relatively few papers focus on issues related to dynamic voltage scaling. Most of the previous work focuses on reducing the power consumption of the computer by turning off devices and the processor when the system is idle. While most of these ideas are also applicable to DVS processors, they do not address how the optimum performance level of the processor can be found. The most important papers about power management in general are summarized in Section 2.2.1. We address the previous work related to dynamic voltage scaling in Section 2.2.2 and summarize the various assumptions about processors supporting dynamic voltage scaling in Section 2.2.3.

2.2.1 System level

A good overview of issues related to energy management of the entire system can be found in [26]. In [8], Culbert details the design of the Apple MessagePad PDA with particular attention to all the energy saving details that were necessary to meet the project’s design goals. The goal of the project was to design a portable handheld computer that could last a week on batteries with typical use, even while running processor intensive applications (at the time) like handwriting recognition. Instead of retrofitting an existing system for power-efficient operation, the designers could contemplate many trade-offs for energy savings that would be unavailable to designers of conventional desktop computers.

Many people have observed that computers spend much of their time idle. The work by Golding et al. provides a taxonomy of idle-time detection and prediction algorithms and identifies metrics for evaluating these algorithms [16]. The main idea discussed in the paper is to schedule useful maintenance tasks when the system would otherwise be idle. Three idle tasks are evaluated: disk power-down to save energy, delayed writeback, and eager segment cleaning to improve disk performance. The authors conclude that even simple predictors work well for predicting idle periods. In the case of the power-down idle task, as much as 70% of the power could be saved.

In [15] and [35], Noble et al. argue that a collaborative relationship between the operating system and applications can be used to meet user-specified goals for battery duration. A user of a mobile computer, for example, can specify that he wants to use the computer for five hours, and then the Odyssey environment coordinates applications to reduce their power consumption accordingly. Their approach takes advantage of the fact that applications can often dynamically modify their behavior to save energy. For example, a video player could lower the frame rate or picture quality and even switch to black-and-white output to reduce power consumption. The authors do not investigate dynamic voltage scaling; they assume that to save energy, one must reduce the amount of work done instead of running it at a lower rate. Also, it is not clear from the paper how their technique would apply to interactive applications where it is more difficult to discard unnecessary work.

In [4], Benini et al. describe a low-overhead workload monitoring system that collects data on the usage of system resources (disks, CPU, keyboard, and mouse). The monitoring technique aids a power manager to put unneeded resources into a low-power sleep state. To do this operation without undue performance impact, the power manager must take the time it takes to wake up a sleeping component into account. The authors describe an API implemented in the Linux kernel that facilitates communications between power-managed devices and the power manager. While the paper gives interesting data about the statistical behavior of managed resources, it is unclear how effectively the use of that information reduces power consumption.

Lorch et al. attempt to improve the power management scheme in the MacOS, which is based on an inactivity timer, by designing a power management scheme that turns off the processor when there is no runnable process [30]. On the MacOS, this strategy is not as straightforward as it might seem since processes can be scheduled and run even when they have no useful work to do. The problem is that originally the MacOS was designed to be a single-user operating system, without preemptive multitasking, where the running application is allowed to have full control of the machine. Applications were not always designed to gracefully cooperate with each other and often programmers made the assumption that their application was the only important one in the system. To work around these problems, the authors propose a modified scheduler and heuristics to tell when an application is doing no useful work (e.g. spinning in an idle loop). These changes enable to up to a 66% savings in processor energy, and a 20% corresponding increase in battery life of the total system.

2.2.2 Dynamic voltage scaling

While many of the papers in the previous section address the reduction of energy consumption of the processor, their usual solution is to put the processor into idle mode when nothing is executing on it. However, a more fruitful way of saving energy is to also adjust the performance level (along with the voltage) of the processor to the applications' requirements during execution. The goal is to finish tasks exactly on their deadlines. Going too fast wastes energy, going too slow deteriorates the user experience and could cause critical failures in real-time systems.

In the context of real-time systems, researchers have explored voltage and frequency scaling as a means of reducing power consumption. Papers [36][24][44] present algorithms and theoretical models that allow one to incorporate voltage scheduling into real-time schedulers. However, these papers are not directly applicable to general-purpose operating systems, since the workloads are expected to have well defined characteristics (periodicity, resource requirements, deadlines, etc.). Moreover, the user must explicitly specify these characteristics to the scheduler.

Our research is more closely related to the work described in [40][48][17], where performance-setting decisions are made automatically guided by the ratio of idle time to busy time on the processor. Unfortunately, existing approaches are not very accurate and can be easily confused by irregular processor utilization. We improve on these algorithms by using task-level information from the OS kernel. A more detailed summary of the directly related work follows below:

In [48], Weiser et al., sketched out the main ideas behind automated performance scaling. Their mechanism uses the amount of idle time as the guide for finding the optimum level of performance. The practical policy proposed in this paper is called PAST. In this policy the utilization for the most recent interval is computed and if it is above a certain threshold, performance is increased but if the interval includes mostly idle time, performance is reduced.

While the PAST algorithm looks very attractive due to its simplicity and effectiveness on some benchmarks, in [17], Govil et al. point out its shortcomings and propose improvements. One of their complaints is that PAST looks back only at a single interval and thus it smoothes speed poorly: the algorithm keeps on changing performance levels without coming to a steady state thus missing out on opportunities to save power. To remedy the situation, the authors propose a number of algorithms that use varying amounts of prediction, to improve their accuracy. They conclude that smoothing, rather than sophisticated prediction might be the most effective. Accordingly, they propose an algorithm, PEAK, that looks for recurring patterns of processor utilization, with special attention to short bursts of high utilization, and attempts to set processor speed accordingly.

Pering et al. evaluate interval-based voltage scaling algorithms for use on a handheld device [40]. One of the key contributions of this paper is the use of the clipped-delay metric, which takes into account that the length of some events can be increased without affecting the user. In practice, the effectiveness of this technique depends on the allowed increase in delay. The algorithm may degrade performance slightly but yield significant power reduction. While the algorithms used in this paper worked, their performance fell short of optimal. Moreover, the algorithms used some specific knowledge about the executing programs, which may be impractical on a production system. Pering et al. show that while interval-based voltage scaling algorithms work well on benchmarks with regular processor utilization (such as audio playback), they do not fare well on irregular workloads, such as interactive workloads or video playback.

These conclusions are corroborated by Grunwald et al. [18]. They also find that using a weighted average of processor utilization as a guide to future utilization (the AVG_N policy) does not yield the clock speed that would maximize processor utilization. Another problem with this algorithm is that the requirement to average N intervals introduces a delay in responding to processor demand. The authors find that existing heuristics did not fare as well on an actual implementation as previous studies had suggested.

The first commercial microprocessor that takes advantage of dynamic voltage scaling is the Transmeta Crusoe [25]. This processor is also unique in that on top of a

VLIW core, there is a software layer that translates between the processor's external instruction set (x86) and its internal instructions. This software layer includes a dynamic compiler that applies optimizations to executing programs on the fly and also supports a power management mechanism to reduce the processor's operating frequency and voltage. Since the translation software and power manager sit under the operating system and are hidden from it, this arrangement ensures that preexisting software can take advantage of the processor's features. Just as in the interval-based performance-setting schemes described above, software detects the ratio between idle and utilized episodes and sets performance-levels accordingly. This fact makes it unclear how well their power-management strategy works on interactive applications.

An interesting twist on dynamic frequency and voltage scheduling is described in [21], where instead of relying on the operating system exclusively for performance-setting, the authors propose to use a compiler. Their model relies on computing the balance between CPU- and memory-intensive operations. If a workload is memory-bound, the processor's performance and voltage levels can be reduced with negligible performance impact. The authors apply advanced compiler optimizations on computation kernels and evaluate their impact on power savings.

2.2.3 Processors supporting dynamic voltage scaling

In [39] and [40], Pering et al. describe the experimental lpARM processor, which was explicitly designed to support dynamic voltage scaling. This processor is based on the ARM8 core and is projected to consume 1.8mW at 8Mhz@1.1V and 220mW at 100Mhz@3.3V. This chip is targeted at ultra-low power consumption and moderate performance. The authors contrast their design decisions with the contemporary StrongARM [22] design, which targets high performance and moderate power consumption. The lpARM shows on order-of-magnitude improvement over the idle-mode power consumption of the StrongARM: the StrongARM consumes 20mW, while the lpARM consumes only about a tenth of that amount. On the lpARM, it takes about 25 microseconds for a complete a 10Mhz to 100Mhz frequency transition with corresponding change of voltage levels. Unlike other processors that support dynamic voltage scaling, the lpARM can continue executing while its frequency is being changed. Other processors usually pause for about 20 microseconds until the frequency is set at the target level and then continue executing while the voltage is changed to match the frequency (note that if the performance is increased, then the voltage is set to the higher level before changing the frequency).

A system based on the StrongARM-1100 processor that uses voltage and frequency scaling is described in [41]. While their processor was not explicitly designed to support dynamic voltage scaling, the authors find that with the addition of voltage regulators, the processor can operate between 59Mhz@0.79V and 251Mhz@1.65V. This processor consumes 105.8mW at 59Mhz and 963.7mW at 251Mhz. Power consumption is much higher than the lpARM but the StrongARM also enables higher peak performance. The authors report that it takes 140 microseconds to perform a frequency change. This duration is insensitive to the frequency and is needed to resynchronize the PLL. Upward voltage changes occur even more rapidly: it takes only 40 microseconds to change the voltage from minimum to maximum value. Downward voltage changes, on the other hand take about 5.5 milliseconds, since the capacitance of the voltage regulator needs to be discharged. These results confirm that fast performance and voltage changes can be reasonable expected on processors that are explicitly designed to support dynamic voltage scaling.

The Transmeta Crusoe 5400 [25] operates between 300Mhz@1.2V and 644Mhz@1.6V. A full voltage transition takes less than 300 microseconds. There is a pause of about 20 microseconds before the frequency is changed. The voltage range and thus the efficiency of the SA-1100 is much larger than that of the Transmeta Crusoe.

In our performance-setting studies, we evaluate the effectiveness of our designs based on assumptions about the Intel XScale microarchitecture [23]. These assumptions are detailed in Section 6.4.1. In our most realistic model, we assume that the processor operates between 150Mhz@0.75V and 773Mhz@1.5V and that it takes 1 millisecond to change the voltage levels, along with 20 microseconds for a frequency change. We believe that these are conservative assumptions and that as processor are optimized for dynamic voltage scaling, these latencies will be reduced or completely eliminated.

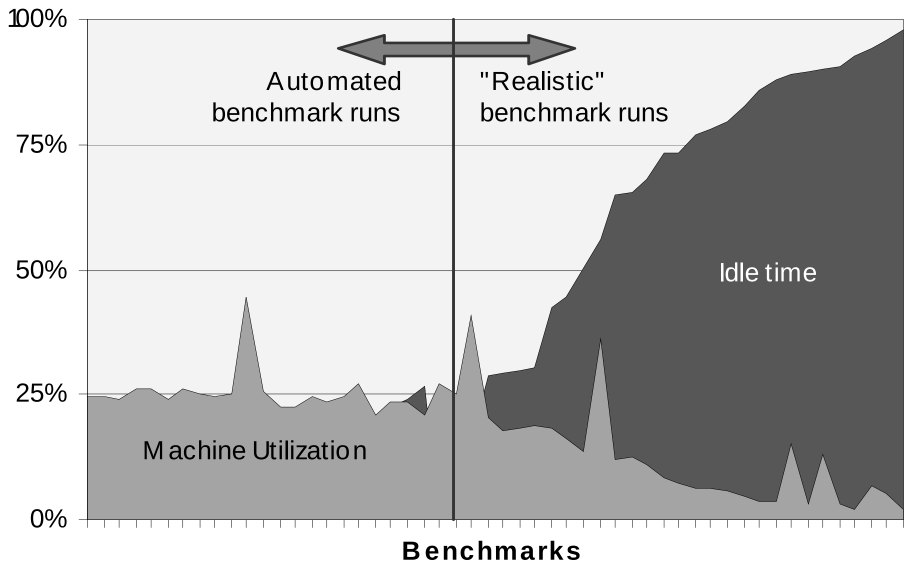
Chapter 3
Thread-level parallelism

In this chapter we present our metrics and the initial results of our investigations into the characteristics of thread-level parallelism of desktop applications. These findings motivate the need for a mechanism that can focus on important episodes during the execution of the program, instead of looking at the execution of benchmarks in the aggregate. We examine a wide variety of existing desktop workloads (over 50) across three different operating systems (Windows NT, BeOS and Linux) and quantify the amount and nature of thread-level parallelism (TLP) in the system. Our results show that OS and application structure have a significant effect on TLP. While most of the workloads exhibit only moderate amounts of parallelism (under 1.5), there is evidence that shows that many of these workloads are not inherently single threaded.

3.1 Introduction

Processor architects are increasingly looking at support for multithreaded workloads. A new generation of processors such as the IBM Power4 [9], Compaq EV8 [10] and Sun MAJC [32] are using simultaneous multithreading [47] or single chip multiprocessing [19] to achieve high performance. While multiprocessing is already prevalent in servers, where multiple clients present an obvious source of parallelism, the case for multiprocessing is not as clear for desktop applications. Moreover, desktop applications are difficult to simulate on architectural simulators due to user interactions and the graphical user interface. For these reasons, researchers have often resorted to concurrently running multiple independent workloads to simulate these workloads. While this methodology is sufficient for certain studies, it has little correlation to real usage models.

Over the past years, multiprocessors have moved from the server segment to workstation users and are now entering the desktop arena as well. Recently, Apple Computer made dual PowerPC based machines standard across most of its desktop PowerMac line [2]. Moreover, processors capable of executing multiple instruction streams concurrently



The figure shows machine utilization and idle time of 52 workloads on a quad-processor machine. The automated benchmarks were driven by a GUI automation tool (e.g., Visual Test), while the realistic workloads were run by a human.

Figure 3.1: Machine utilization and idle time of 52 workloads

will be readily available in the near future. SMT- and CMP-based products have already been announced [9][10][32] and the cost of existing multichip multiprocessors have been decreasing steadily [2].

Figure 3.1 illustrates the machine utilization and idle time characteristics of 52 desktop workloads across three operating systems (Windows NT, BeOS, and Linux) running on a quad-processor machine. Machine utilization is a measure of how effectively a machine's computing resources are exploited and is 100% if all processors in the system are utilized. It would be difficult to make a case for multiprocessing for desktop workloads based on the presented machine utilization data. Very few of the workloads exceed 25% machine utilization, suggesting that for the most part, only one out of the four available processors is exercised. Some of the benchmarks exhibit even lower utilization in the 5% to 10% range, suggesting that a single processor is more than powerful enough for running these workloads. What is the need for multiprocessing when a single processor seems to be adequate?

The problem with the machine utilization metric is that it weighs all parts of the benchmark's execution equally. From the metric's point of view, generating a page of output on the screen is as important as the idle period when the user is consuming the data (think time) and the processor is doing no useful work. We define idle time as the percentage of time all processors in the machine are idle simultaneously. Machine utilization can only be used to accurately measure the effects of concurrent execution if idle time during the benchmark run is close to zero.

As the figure shows, the amount of idle time in desktop applications can be very large. In some cases idle time amounts to more than 90% of total execution time. This high ratio should not be unexpected since interactive applications run at the rate at which the user interacts with them, which is determined by human cognition and motor skills and includes significant think time. The high proportion of idle time can be obscured by the use of automated benchmarks, such as Sysmark 98 [46] or Winstone 99 [49], that perform each operation as soon as the previous operation completes without taking think time into account. The automated benchmarks in Figure 3.1 have an average of 12% idle time versus 64% for the realistic benchmark runs.

To get around the limitations of the machine utilization metric, we define a new metric called thread-level parallelism (TLP). TLP is the machine utilization over the non-idle portions of the benchmark's execution. This definition sidesteps the problems of the original metric and allows us to use more realistic desktop workloads that can include a lot of idle time. Intuitively, TLP is a metric of speedup due to concurrent execution on the non-idle portions of the workload.

Our goal is to gain insight into the amount and nature of thread-level parallelism in a wide variety of existing desktop applications and to speculate on what can be expected in the near future. We are not only interested in the number of threads that are spawned during the run of a benchmark, but also whether these threads actually run concurrently. To this end, we have developed a methodology (described in Section 3.2) for measuring thread-level parallelism (TLP) that is portable across operating systems and have measured a large number of desktop applications (over 50) under Windows NT, Linux and BeOS running on a multiprocessor machine. Our data collection methodology hooks into the kernel at a very low level, which allows us to see the activities of all threads in the system and takes all the synchronization and scheduling overhead into account. The collected data can also be used to estimate the TLP of machine configurations with fewer processors than the one on which data has been collected (Section 3.2.2). Some of the questions that we seek to answer are the following:

- To what degree are existing applications threaded?
- How much does the OS contribute to TLP?
- How do modern run-time environments affect TLP?

The chosen operating systems allow us to look at a large number and variety of applications. Windows NT (Section 3.4.1) gave us a platform that runs many of the existing Windows applications on multiprocessing hardware (Windows 98, currently a more prevalent desktop OS does not support multiprocessing) as well as a platform for looking at Java applications. We are interested in how the language-level concurrency support of Java applications and their modern run-time environments influence TLP. BeOS (Section 3.4.2) is a heavily threaded modern operating system, intended for desktop users. Since the operating system has supported SMP hardware from its early days and subsequently its applications have been written with multiprocessing in mind, we looked at it as an indicator of what can be reasonably expected in desktop applications in the near future. Unlike in the previous operating systems, the use of threads is not as pervasive in the Linux 2.2 kernel (Section 3.4.3). While the operating system supports the use of kernel threads (which are just regular processes) through the pthreads [28] interface, most applications are not multithreaded.

3.2 Methodology

3.2.1 Trace collection

Our goals for the measurement process were to come up with a portable technique that is minimally intrusive, works with any unmodified workload, and whose results can be processed off-line to generate interesting metrics and simulate different machine configurations. Instead of using a small number of automated workloads (where the run of the benchmark has been made to be reproducible), we wanted to examine a large cross-section of applications under realistic usage conditions.

The core of our methodology relies on invoking a measurement hook every time a thread starts executing on any of the processors on a multiprocessor system. Our benchmark machine is a four-way multiprocessor, with 500Mhz Pentium III processors and 512MB of main memory. When the hook is invoked, our function records a timestamp, the CPU number on which the switch occurred and the thread ID of the thread that was swapped in. This information is added to a trace and is saved to disk at the end of the observation period. The trace can then be processed off-line to derive TLP information accurately, since the timestamp counters on all processors are synchronized when the machine is booted. To make our traces more complete, we have added information about thread and process creation and deletion, process names and thread-process relationships (i.e., which threads belong to which processes).

The Windows NT implementation uses callbacks to a device driver to record all aforementioned information, while a user-level process controls the trace acquisition. Our benchmarks were run under Windows NT 4 Server (SP5) operating system, since Windows NT Workstation only supports a machine with up to 2 processors. One of the differences between the two operating systems is the time quantum that is given to foreground and background applications. We collected data with different time quantum settings, however it does not have a significant bearing on the TLP numbers.

On all platforms the driver has very low overhead, since it uses no locks and does very little computation. Trace events belonging to each CPU are saved in separate buffers so that the driver can be invoked from each CPU concurrently without using locks. The Linux (kernel 2.2.3) implementation differs from the one for Windows NT in that instead of writing a device driver, we inserted the required calls into the kernel directly and added two system calls to control the measurement process.

While we use the same post-processor on the BeOS as on the other operating systems, trace collection is done using an existing facility in the OS for recording scheduler events. While this approach has certain limitations, we could not find any way of directly hooking the desired thread events. The downside of using the scheduler event log is three-fold: not every piece of information is preserved in the trace, it only works with at most a dual processor machine and the size of the collected trace is limited. To figure out process names, and thread-process relationships, we implemented a user-level program that periodically wakes up and records this information. To avoid disturbing the results, we do not collect thread creation and destruction times and only look at the state of threads in processes that are significant for the workload. This monitor process inflates our TLP numbers by at most 0.02. Using two processors instead of four is not a great limitation, since most desktop applications exhibit a TLP of less than 2. The size of the collected trace is limited to 240K, due to a built-in buffer size.

3.2.2 Metrics

The data collected for each workload is summarized by specifying the total execution time, machine utilization (MU), thread-level parallelism (TLP) numbers and information about the threads and processes that were active during the run of the benchmark.

The TLP data is summarized in four fields. The first field shows what percentage of total execution time (c_i) that exactly $i = 0, ..., n$ (the number of CPUs in the machine) threads are executed concurrently. Correspondingly, c_0 specifies the amount of idle time experienced by the benchmark. The second field shows the measured amount of thread-level parallelism of the benchmark using four processors and the expected amount of TLP on a dual-processor machine. The third and fourth fields describe what percentage of the concurrency comes from within a single process or multiple processes. When possible, processes are classified into two categories: application or system processes, depending on whether they are spawned by the running application or the operating system.

Whenever possible, we show data about the number of active threads and processes in the machine as well as their lifetimes. A thread or process is active if it was scheduled to run at least once during the run of the benchmark. The thread and process lifetime is given as a percentage of the runtime of the benchmark.

The data in the first parallelism field is used to derive machine utilization and TLP. Machine utilization is a measure of how well the processing resources are taken advantage of during the execution of the benchmark. Equation 2 shows how machine utilization is derived from the collected data. Machine utilization is 1 if all processors in the machine were fully utilized.

$$MU = \frac{\sum_{i=1}^{n} c_i i}{n} \qquad \text{(EQ 2)}$$

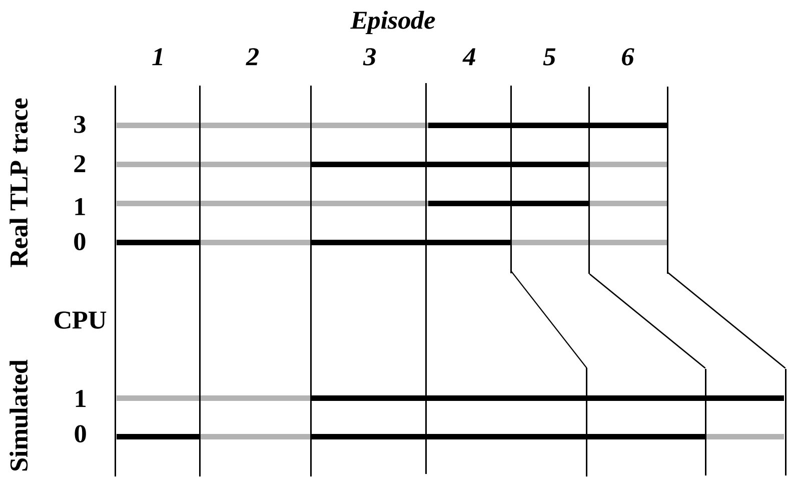


Figure 3.2: Scaling TLP results to a dual processor machine

The problem with this metric is that if the benchmark incurs a significant amount of idle time (due to I/O activity, the user interface or periodic activity), its value can be misleading. For this reason, we have defined *TLP* (Equation 3) as the number of concurrent threads that are executing on the machine if at least one non-idle thread is executing.

$$TLP_n = \frac{\sum_{i=1}^{n} c_i i}{1 - c_0} \qquad \text{(EQ 3)}$$

TLP is between 1 and *n* (the number of processors in the machine). It is important to recognize that the parallelism metric does not necessarily mean that given *n* processors, the workload will get done *TLP* times faster, it only states that the non-idle (processor bound) portions of the program will benefit from the specified speedup. We would have liked to separate idle time due to I/O, however we have been unable to find a portable and reliable way of doing this.

Knowing how the concurrency of an application would scale up to more processors requires information about thread synchronization, however our intuition was that scaling down is a simpler problem. If four threads could execute concurrently on a quad processor machine, then the same threads would also fully utilize a machine with fewer number of processors, but would execute for a longer time. We make the conservative assumption that there is a join point between parallel episodes of different cardinality.

Benchmark	Expected 2 processors	Actual 2 processors	 4 processors
Elastic Reality - Sys98	1.07	1.07	1.07
Photoshop - Sys98	1.15	1.12	1.23
PowerPoint - Sys98	1.07	1.07	1.07
SoundForge - Wins99	1.12	1.12	1.14
Visual C++ - Wins99/MP	1.75	1.74	1.98

Table 3.1: Concurrency on a dual processor vs. a quad-processor machine

This means that when scaling down the workload, parallel episodes with higher TLPs cannot be overlaid with less parallel ones; they must complete before moving on to the next episode. Figure 3.2 illustrates a hypothetical example where data from a quad-processor is scaled to a dual-processor machine. Note that episodes 4 and 5 have an increased run-time on the simulated machine due to the above stated assumptions.

Equation 4 is an elaboration of Equation 3 and can be used to estimate the expected TLP of the measured benchmark on computers with fewer processors than on the measurement machine. In the equation *k* specifies the reduced number of processors and *n* is the number of processors for which the data has been collected (*n* > 1 and 1 <= *k* <= *n*).

$$TLP_k = \frac{\sum_{i=1}^{n} c_i i}{\sum_{i=1}^{k} c_i + \frac{1}{k} \sum_{i=k+1}^{n} c_i i} \qquad \text{(EQ 4)}$$

To check the validity of our assumptions, we reran the subset of our benchmarks that were completely automated on the same machine with two processors turned off. Table 3.1 shows the measured values along with the expected values predicted by Equation 4. As can be seen, this simple formula can be successfully used to predict the TLP of the applications on a dual processor machine. The values for PhotoShop show the greatest difference between predicted and actual values. In this benchmark there is a significant reduction of dead-time (c_0), which causes the overall proportion of c_1 to increase and c_2 to decrease.

3.3 Benchmarks

We chose Windows NT Server 4.0 (SP5) as our primary platform for these investigations, since it runs a large variety of applications, has robust support for threading and can run on multiprocessor hardware. The Sysmark 98 [46] and Winstone 99 [49] benchmark sets provided a large number of applications for our initial observations. The advantage of these benchmark suites is that they consist of commercial applications that are driven by a GUI automation tool to achieve life-like behaviour. The reproducibility of the runs of these benchmarks has allowed us to fine-tune our methodology and to make comparisons between runs on different computers. However, running only fully-automated benchmarks would have obscured the fact that many of the applications incur a lot of idle-time when running in interactive mode.

Java applications were run on Windows NT using the Java 1.3beta SDK with the HotSpot Client VM. The BeOS applications were run on BeOS 4.5.2 and we used the RedHat 6.0 distribution with our modified 2.2.3 kernel for Linux benchmarks.

3.4 Results

3.4.1 Windows NT

Most Windows NT desktop applications exhibit only moderate amounts of TLP (see Table 3.2). The highest numbers come from multimedia applications (both authoring and playback), program development tools as well as from scientific and engineering applications. The only applications that achieve significantly more TLP than 1.3 are applications that are either hand parallelized (Parallel MPEG player) or that have “obvious” areas of parallelism such as Visual C++, where projects are compiled concurrently. However, even this application misses out on some opportunity for concurrency, since files are not compiled in parallel, only projects. When compiling a single project the TLP is 1.18, compiling two projects it is 1.98. The gnu make and gcc based compilation environment on the other hand achieves a concurrency of 3.27 when compiling a single project.

Category	Application	Workload	MU	Thread-level Parallelism (TLP)											
				Threads							Intra process		Inter process		
				c_0	c_1	c_2	c_3	c_4	TLP_4	TLP_2	App	Sys	App-App	App-Sys	Sys-Sys
Database	Cloudscape - Java[a]	embedded	18%	29.7%	68.2%	1.7%	0.3%	0.1%	1.04	1.03	22%	0%	47%	30%	1%
		rmijdbc 1c	24%	11.8%	81.0%	5.4%	1.4%	0.4%	1.11	1.09	12%	0%	47%	38%	2%
		rmijdbc 2c	37%	7.1%	45.5%	41.4%	4.2%	1.7%	1.59	1.53	16%	0%	49%	34%	2%
		rmijdbc 4c	52%	8.6%	39.0%	13.2%	16.1%	23.1%	2.26	1.68	19%	0%	56%	24%	1%
	Paradox 8.0	Sys98	21%	19.2%	77.3%	3.1%	0.3%	0.1%	1.05	1.05	17%	0%	28%	54%	1%
Design	Bryce 2	Sys98	25%	1.9%	97.2%	0.9%	0%	0%	1.01	1.01	0%	0%	63%	37%	0%
	CorelDRAW 8.0	Sys98	25%	4.4%	92.7%	2.7%	0.1%	0%	1.03	1.03	0%	0%	14%	86%	0%
	Elastic Reality 3.1	Sys98	25%	7.3%	86.3%	6.0%	0.3%	0.1%	1.07	1.07	0%	0%	32%	67%	0%
	Extreme 3D	Sys98	24%	4.6%	93.7%	1.6%	0.1%	0%	1.02	1.02	0%	0%	32%	68%	0%
	Image/J - Java		18%	42.2%	44.4%	12.4%	0.8%	0.2%	1.25	1.24	64%	0%	7%	29%	0%
	Photoshop 4.0.1	Sys98	23%	24.2%	68.4%	2.0%	0.9%	4.5%	1.23	1.15	1%	0%	2%	97%	0%
		Win99[b]	22%	52.5%	31.2%	3.6%	1.6%	11.0%	1.84	1.47	28%	0%	36%	31%	5%
		Win99/MP	35%	41.5%	28.5%	4.0%	2.1%	23.9%	2.36	1.66	50%	0%	28%	22%	0%
Dev.	Visual C++ 5.0	Win99	25%	16.0%	72.4%	9.3%	1.4%	0.9%	1.18	1.15	0%	1%	31%	68%	0%
		Win99/MP	44%	10.4%	24.9%	46.8%	13.3%	4.7%	1.98	1.75	0%	1%	49%	49%	0%
	GNU make / g++[c]	gzip	36%	55.8%	8.8%	2.1%	1.5%	31.7%	3.27	1.88	0%	0%	64%	37%	0%
G	Quake 2		24%	5.3%	92.8%	1.8%	0.2%	0.0%	1.02	1.02	2%	0%	15%	82%	0%
Internet	Frontpage 98	Win99	9%	70.0%	25.0%	3.4%	0.9%	0.7%	1.24	1.20	1%	1%	56%	43%	0%
	HotJava - Java	Browsing	6%	78.0%	19.1%	2.3%	0.5%	0.2%	1.17	1.15	76%	0%	5%	18%	0%
	Internet Explorer 5	Misc	3%	89.5%	8.5%	1.5%	0.3%	0.2%	1.25	1.21	23%	1%	31%	44%	0%
	Netscape 4.05	Sys98	25%	17.8%	67.6%	12.8%	1.7%	0.1%	1.20	1.19	17%	0%	19%	63%	1%
Authoring	Premiere 4.2	Sys98	25%	5.8%	89.2%	4.7%	0.3%	0.1%	1.06	1.06	65%	0%	12%	22%	0%
		Win99	26%	13.0%	72.0%	14.2%	0.7%	0.1%	1.18	1.18	0%	0%	19%	81%	0%
	SoundForge 4.0	Win99	21%	26.9%	64.4%	7.5%	1.0%	0.1%	1.14	1.13	0%	7%	15%	78%	1%
	MPEG Encoder	Sys98	22%	15.9%	79.7%	4.0%	0.3%	0.1%	1.06	1.06	13%	0%	45%	42%	0%
Multimedia playback	Parallel MPEG	HDTV	94%	0.0%	4.4%	2.4%	5.8%	87.4%	3.76	1.98	98%	0%	1%	1%	0%
	QuickTime 4.0.3	MP3	4%	85.9%	13.1%	0.9%	0.1%	0.0%	1.07	1.07	17%	0%	25%	57%	0%
		QT	16%	44.4%	46.2%	8.9%	0.5%	0.0%	1.18	1.17	78%	0%	10%	12%	0%
		QT - 2[d]	25%	18.5%	64.9%	14.8%	1.7%	0.2%	1.23	1.21	43%	0%	29%	27%	0%
	RealJukebox	MP3	4%	87.9%	10.2%	1.6%	0.3%	0.1%	1.20	1.18	5%	0%	32%	62%	0%
	Windows Media Player	AVI-local	6%	76.9%	21.0%	1.8%	0.3%	0.1%	1.11	1.10	54%	0%	27%	19%	0%
		AVI-net	6%	79.6%	17.7%	2.0%	0.5%	0.1%	1.17	1.15	20%	1%	22%	56%	0%
		AVI-crypt	8%	73.5%	23.2%	2.9%	0.4%	0.0%	1.14	1.13	37%	2%	28%	33%	0%
		MP3	2%	92.8%	6.6%	0.5%	0.1%	0.0%	1.09	1.09	18%	0%	49%	32%	0%
Lang	NatSpeaking 2.02	Sys98	24%	5.0%	93.0%	1.7%	0.3%	0.0%	1.02	1.02	33%	0%	30%	37%	0%
	PowerTranslator		18%	29.4%	69.9%	0.5%	0.1%	0.0%	1.01	1.01	0%	0%	71%	29%	0%
Productivity	Excel 97	Sys98	26%	5.5%	85.6%	8.5%	0.4%	0.0%	1.10	1.10	3%	0%	4%	93%	0%
	Lotus SmartSuite	Win99	14%	48.6%	46.8%	3.8%	0.7%	0.1%	1.11	1.10	2%	1%	8%	88%	0%
	Microsoft Office[e]	Win99	23%	22.4%	63.0%	13.1%	1.3%	0.2%	1.21	1.20	2%	0%	74%	23%	0%
	PowerPoint 97	Sys98	25%	7.2%	86.6%	5.6%	0.4%	0.0%	1.07	1.07	3%	0%	10%	83%	3%
	Word 97	Sys98	26%	4.8%	85.2%	9.7%	0.3%	0.0%	1.11	1.11	0%	0%	21%	77%	1%
	OmniPage Pro 8.0	Sys98	22%	14.9%	81.6%	3.0%	0.4%	0.1%	1.05	1.04	0%	0%	67%	33%	0%
Sci/Eng	AVS Express 3.4	Win99	26%	4.7%	86.7%	8.4%	0.2%	0.0%	1.09	1.09	0%	0%	28%	72%	0%
	JSV - Java[f]		25%	7.8%	84.9%	5.9%	1.0%	0.4%	1.10	1.09	56%	0%	8%	36%	0%
	MicroStation SE	Win99	24%	17.7%	70.9%	10.9%	0.4%	0.0%	1.15	1.14	10%	0%	26%	64%	0%
		Win99/MP	27%	18.4%	55.0%	26.2%	0.4%	0.1%	1.33	1.33	72%	0%	2%	26%	0%

a.An object database system. Modes: embedded (server and client in single program) and client-server (rmijdbc) with 1, 2 and 4 clients.
b.This benchmark did not complete, due to a bug in the Winstone 99 benchmark driver.
c.make -j 8 run on the gzip sources under cygwin (sourceware.cygnus.com/cygwin).
d.Playing back two QuickTime movie streams concurrently (Sorensen compression, 640x288, millions of color).
e.The data presented here is for the middle portion of the run, since the entire trace was too big for the post-processor.
f.Scientific Visualization. Displaying and rotating various molecules.

Table 3.2: Windows NT benchmark results

In their study of the characteristics of desktop applications [27], Lee et al. observed that most of the instructions are executed from a single dominant thread. The TLP numbers in our study confirm this observation, however it is worth noting that results can vary widely based on the workload. For example when running Adobe Photoshop, we observed that our TLP numbers varied from 1.23 to 2.36 implying that a single thread did not always dominate during execution. However, Lee et al. reported that during their run 97.16% of instructions came from the primary thread. While the two metrics are not directly comparable, it is unlikely that one could observe an average of two concurrent threads during execution, while executing nearly all instructions from only one of them.

The amount of concurrency can fluctuate greatly when running the same application with different workloads. An example of this is Photoshop, which has a concurrency of 1.23 when running as part of Sysmark 98, and 2.36 under the Winstone 99/MP workload.

Java applications do not seem to exhibit significantly more TLP than their traditional counterparts. It is true that they use more threads than other applications on average, however these threads tend not to run concurrently. The pervasive use of threads in Java does translate into a large percentage of the concurrency coming from within the application process (50% to 80% in our benchmark). This number includes both threads from the Java application and the Java VM.

The database workloads in our measurements do not show a significant amount of concurrency as long as there is only a single client accessing the data. However, if two clients are used with the Cloudscape database, the TLP jumps to 1.59. This usage scenario could become realistic on the desktop as loosely coupled object based environments (such as Jini and JavaSpaces) become more prevalent.

The video playback applications in our benchmark suite all exhibit different kinds of concurrency characteristics. The Parallel MPEG player has been hand parallelized to play back an HDTV video stream. The existing hardware is barely enough to accomplish this task (TLP is 3.76). The other playback applications have not been parallelized to nearly the same degree. QuickTime uses multiple threads to decode and play back the multiple streams of multimedia information in a movie: it spawns a thread for audio and

Benchmark	TLP_4	App-Sys TLP	Thread from operating system process		
			1.	2.	3.
Internet Explorer 5	1.25	44%	System[a] (85.55%)	UI (6.78%)	Win32 (5.39%)
JSV - Java	1.10	36%	System (83.74%)	Win32 (13.92%)	UI (1.06%)
Microstation SE	1.15	64%	System (71.86%)	Win32 (26.43%)	Services (1.33%)
Netscape 4.05	1.20	63%	System (84.30%)	Win32 (10.74%)	UI (3.99%)
PhotoShop 4.0.1	1.23	97%	System (98.5%)	Win32 (1.37%)	UI (0.10%)
PowerPoint 97	1.07	83%	System (75.64%)	Spooler (11.47%)	Win32 (6.37%)
RealJukebox - MP3	1.20	62%	System (77.19%)	Win32 (11.97%)	Services (8.67%)
Visual C++ 5.0	1.18	68%	System (84.48%)	Win32 (14.90%)	Services (0.24%)

a.The System process contains kernel and device driver threads, UI stands for Explorer.exe, which is responsible for the Windows NT user interface, Win32 is responsible for handling the Win32 API calls, Services is the service controller (starts, stops and controls service daemons) and Spooler (spoolss.exe) is the spooler service daemon.

Table 3.3: System processes contributing to inter-application TLP

one for video (depending on the movie, there could be more or less threads). However, neither one of these threads is very resource intensive and the audio decoder has about 20% of the processing requirements of the video thread (based on its 1.18 TLP, 78% of which is intra application concurrency). The machine idles 44% of the time during play-back, since once it reaches the desired service quality (the highest in our case), it has to ensure that frames are presented at a uniform rate. The Windows Media Player has similar characteristics; however, it was only able to play our AVI movie at a low framerate. Dis-playing two video streams concurrently estimates the video workload of a high end video conferencing software. The measured TLP of two high-quality QuickTime movie streams increases to 1.23 and includes a significant reduction of idle time (18.5% of total execu-tion time).

When interactive workloads are run under realistic conditions, the system is idle a majority of the time. This fact is obscured by the automated benchmarks; a case in point is the comparison of Netscape from the Sysmark 98 benchmark suite to Internet Explorer and HotJava, both of which were run by hand. While the Netscape benchmark spends only 17.8% of its execution time idling, the other two web browsers wait for something to do about 80% of the time.

VMware allows one to run operating systems inside a virtual machine. To accomplish this task, it has to emulate the behaviour of the entire machine, including all the necessary devices. In our measurements, the VM was a significant source of TLP when device emulation was used heavily, however under most conditions its contribution to TLP was not significant. Moreover since it only emulates a uniprocessor, applications running within the VM are also limited to uniprocessor performance, even if they are multithreaded.

A negligible fraction of concurrency comes from running operating system threads (from one or more processes) in parallel. However, application and system threads running at the same time make up a large portion of overall TLP. The greatest single contribution comes from running an application thread in parallel with a thread from the System process, which contains the device driver and kernel threads. Table 3.3 shows the operating system processes that contribute the most to interprocess TLP between application and system threads. As can be seen from the table, the operating system threads of Windows NT contribute significantly to the concurrency of the platform.

3.4.2 BeOS

Given the BeOS' reputation as an operating system designed with multiprocessing and multithreading in mind, we expected it to yield higher TLP numbers than Windows NT. Overall this expectation turned out to be justified but the range and sophistication of the applications we measured under the BeOS is not nearly as great as under Windows NT. The results are shown in Table 3.4.

Two of the measured applications have exact counterparts under Windows: Abuse 2.0 and Quake2. Quake was a straight port to the BeOS, and any increase in concurrency is due to the OS: the TLP improves from 1.02 to 1.08, not a significant change.

The difference is greater in Abuse. Abuse under Windows runs as a DOS application (unfortunately it crashes under Windows NT), which is effectively single-threaded. However, under BeOS in "normal" mode, the TLP is 1.23. This increase in concurrency is due to the improvements made during the porting process and shows that this workload is not inherently single threaded. Abuse also provides a "fast" mode in which the game attempts to use as much of the available resources as possible without limiting the speed

App	Workload	Total time (s)	Utilization MU	Thread-level Parallelism (TLP)						Threads	
				Threads			TLP$_2$	Process			
				c_0	c_1	c_2		Intra	Inter	Active	Created
Media-Player	AVI	11.94	26.99%	50.5%	45.0%	4.5%	1.09	19.6%	80.4%	44	0
	Quick-Time	11.26	9.70%	82.7%	15.1%	2.1%	1.12	13.4%	86.6%	55	0
3dmov	Pulse -QT	15.06	40.84%	28.7%	60.9%	10.4%	1.15	5.1%	94.9%	44	0
Abuse 2.0	Normal	3.71	16.54%	73.2%	20.6%	6.2%	1.23	25.7%	74.3%	53	0
	Fast	4.74	73.38%	1.4%	50.4%	48.2%	1.49	35.4%	64.6%	57	0
Axia 1.01	Normal	2.89	37.78%	30.5%	63.5%	6.0%	1.09	47.1%	52.9%	32	0
	Fast	3.2	62.63%	1.8%	71.1%	27.1%	1.28	50.9%	49%	41	0
Quake2	Demo	29.79	53.99%	0.2%	91.5%	8.2%	1.08	3.7%	96.3%	52	0
Life	2 threads	14.48	81.51%	15.9%	5.2%	78.9%	1.94	55.2%	44.8%	44	6
	8 threads	8.8	70.56%	25.8%	7.4%	66.9%	1.90	70.9%	29.1%	49	12
Flight		10.6	55.04%	0.0%	89.9%	10.1%	1.10	8.7%	91.3%	42	0
Charts - space	LoFi	12.04	6.61%	90.4%	5.9%	3.6%	1.38	36.1%	63.9%	42	0
	HiFi	15.08	96.89%	0.0%	6.2%	93.8%	1.94	46.2%	53.8%	43	0
NetPositive	Apple	3.05	24.92%	65.2%	19.7%	15.1%	1.43	27.6%	72.4%	100	51
	Be	3.82	21.85%	67.9%	20.5%	11.6%	1.36	24.1%	75.9%	140	86
	Wired	2.21	23.57%	65.1%	22.7%	12.2%	1.35	21.6%	78.4%	73	31

Table 3.4: BeOS benchmark results

of the action to playable levels. The fast mode measurement shows even greater concurrency of 1.49. These TLP numbers might be achievable on a similar game by making the graphics and game logic more sophisticated.

The MediaPlayer benchmarks look very similar to their Windows NT counterparts: the concurrency is low and there is a lot of time when nothing is executing on the machine. This is due to the fact that playback of these streams is tied to a given service quality (frame rate and sound quality) and if they are met, there is simply nothing more to be done. Somewhat of a surprise was the 3dmov benchmark, which includes a 3D rendered moving pulse, onto which a QuickTime movie is projected. While there was a significant decrease in dead time compared to movie playback by itself, the application does not show much increase in concurrency.

The web browsing benchmarks on the other hand exhibit a significant amount of TLP, even when displaying simple web pages. The displayed web pages do not contain

Application	Workload	Total time (s)	MU	Thread-level Parallelism (TLP)						
				Threads						
				c_0	c_1	c_2	c_3	c_4	TLP_4	TLP_2
Gimp	Cubism	24.06	25.27%	0%	99%	1%	0%	0%	1.01	1.01
	Predator	4.54	21.64%	23.5%	68.2%	6.7%	1.4%	0.2%	1.13	1.12
	Weave	15.7	27.04%	2.9%	87.9%	7.6%	1.4%	0.3%	1.11	1.10
Battalion		44.83	25.26%	0%	99%	1%	0%	0%	1.01	1.01
Xanim	AVI	22.6	0.33%	98.7%	1.3%	0%	0%	0%	1.00	1.00
Mozilla M10		29.09	13.19%	49.9%	47.5%	2.6%	0%	0%	1.05	1.05
Make / gcc	kernel src	69.11	82.18%	4.5%	13.2%	3.3%	7.1%	71.9%	3.44	1.92
Netscape 4.5	Apple	20.01	2.4%	91.5%	7.5%	1.1%	0%	0%	1.13	1.13
	Intel	20.01	1.9%	92.9%	6.5%	0.5%	0%	0%	1.08	1.08
	Wired	20.01	4.19%	84.5%	14.4%	1.2%	0%	0%	1.08	1.08
Quake2	Demo	20.01	25.06%	0.1%	99.7%	0.3%	0%	0%	1.00	1.00

Table 3.5: Linux benchmark results

Java applets but simply graphics and text. Compared to similar benchmarks under Windows NT, the BeOS comes out about 0.1 TLP ahead even in the worst case. This is a significant increase and is due to a different rendering approach on the BeOS; every object on the web page is rendered in parallel, as data from the network becomes available. This contrasts with the approach with the other web browsers under Windows NT that do not spawn new threads for objects within a page.

It is not as easy to classify the concurrency of BeOS applications by application and system threads as on Windows NT. The reason is that almost all applications register threads in other processes for handling events and the user interface. As a result, concurrency that would show up as intra process concurrency in NT shows up as inter process concurrency on the BeOS.

3.4.3 Linux

Linux is not a threaded operating system in the same sense as the Windows NT and BeOS. In Linux, the basic unit of execution is a process, however lightweight processes can be created using the clone system call with small overhead, and those can be used as threads. However, the use of kernel threads is not yet pervasive in the operating system and its utilities. Even applications that are threaded do not always use Linux' kernel threads; usually because of unresolved issues regarding the thread safety of standard libraries. For example Netscape Communicator relies on its own user level thread library, which prevents it from taking advantage of multiple processors.

The only workload in our suite that exhibits a significant amount of TLP is the compiler benchmark, which uses parallel make to compile files concurrently. The benchmarks execute three or four threads less frequently than the other two OSs, because there is not as much opportunity to overlay execution of system and application threads.

Unlike Adobe Photoshop on Windows NT, the Gimp image manipulation program does not exploit the thread-level parallelism inherent in its algorithms but derives some concurrency from loading and running plug-ins and the user interface.

The Linux results in this section were obtained on the 2.2 version of the Linux kernel. However, changes in the 2.3 kernel have greatly improved the kernel's ability to take advantage of multiprocessing hardware. Results in future chapters were obtained using the newer kernel.

3.5 Conclusions

Our survey shows that most desktop applications only exhibit moderate amounts of thread-level parallelism. The amount of parallelism can vary greatly between similar kinds of applications and is dependent on the operating system. Applications on the BeOS tend to exhibit higher amounts of TLP than similar applications on Windows NT. Java applications on the other hand, do not exhibit significantly higher degrees of concurrency than their Windows NT counterparts. Most desktop applications incur a large amount of idle time, which can be as much as 90% of the total execution time.

Moore's law predicts a doubling of uniprocessor performance every 18 months. In practice this implies that if there is a workload that runs with TLP of 2 on a dual processor machine today, then in a year and a half there will be a uniprocessor capable of the same feat (to a first order approximation). However, even the more parallel of the existing desktop applications exhibit a concurrency of only around 1.5, which brings the lead time of a dual processor over an equivalent performing uniprocessor to about three quarters of a year.

The amount of concurrency in most of the measured applications is less than two. This suggests that without substantial software changes to expose TLP to hardware, desktop multiprocessors with more than two processors are unwarranted in most situations. The only applications that exhibit higher amounts of concurrency have either obvious sources of concurrency (such as compiling using make -j) or are ones that rely on parallel algorithms at their core (e.g. Parallel MPEG player). While most of the results were collected on a quad processor machine, the expected TLP_2 number shows that in most cases a dual processor machine would exhibit similar amounts of concurrency.

The availability of idle time presents an opportunity to provide a richer software layer for applications. Dynamic profile-feedback based optimizations could be performed during the available time to improve the efficiency of the software.

It is clear from the survey of existing applications that most of the programs use threads as a convenient abstraction for the programmer, not as way to maximize performance. This is partly due to the current prevalence of uniprocessor systems — optimizing for the uncommon multiprocessor machine is not beneficial — and also because programmer time is expensive. Unless writing an application that takes advantage of the parallel hardware is on the same order of difficulty as writing the program in the straightforward way, it is unlikely that we will see a great increase in parallelism. If increased concurrency does not come from the applications, it could come from the platform and the user interface.

The use of soft devices (mostly software implementation of hardware devices, such as modems) could be a significant source of concurrency on multiprocessors. The nature of these applications is to put a constant load on the machine to provide the required signal processing. These types of devices currently include soft modems and soft DSL and DVD players. Unfortunately the current implementations that we know of do not run under Windows NT (most run under Windows 98, which is not a multiprocessor capable OS), so we could not directly measure their contribution. Also, the current implementations usually rely on DPCs (delayed procedure calls) to perform work, which does not show up as thread-level concurrency. However, based on data from Rockwell [7], a soft modem requires about 50MIPS of processing, while a soft DSL would require about 500MIPS. This means that on the measurement hardware, the presence of a soft modem would add approximately 0.1 and the soft DSL about 1 to the concurrency numbers. A theoretical client for video-on-demand services (soft DSL with video playback) could operate with an average concurrency of about 2.

The voice recognition and understanding benchmark are both single-threaded CPU intensive workloads. Even if the applications themselves are not parallelized, they would significantly contribute to the concurrency of the platform if used as part of the user interface. It would mean at least one additional CPU bound thread in addition to anything the user is doing on the machine.

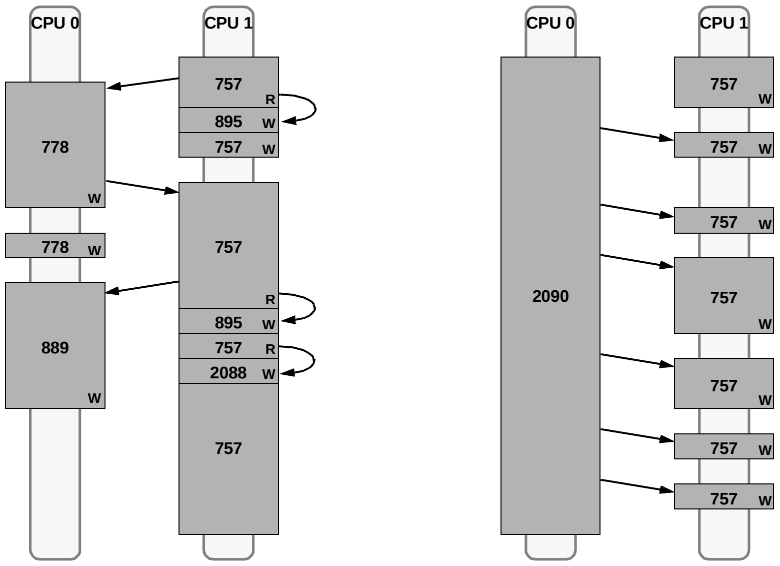
In this section we have focused on the overall thread-level parallelism of our workloads. However, our results indicate that multiprocessing can have a significant impact on the user perceived response time of interactive applications. These issues are addressed in Chapter 5.

Chapter 4
Episode detection and classification

This chapter presents the mechanism for detecting different kinds of episodes during a task's execution. An episode corresponds to a period of system activity with certain clearly discernible characteristics. Currently we distinguish between interactive and periodic episodes. Interactive episodes are initiated by the user (by a mouse click or movement, keyboard interaction, etc.) and last until the computer is done responding to the request. Periodic episodes, on the other hand, are due to recurring tasks running on the computer, such as sound or video playback. During its lifetime, episodes of a task can fall into more than one of these classifications. For example, a music playback process may be part of an interactive episode when it is updating the GUI and a periodic episode when it is decoding music data. The main use of interactive episodes is to automatically measure the user-perceived response times of applications. We use the technique described in this chapter to quantify the impact of multiprocessing on user-perceptible response times (see Chapter 5). The episode detection mechanism also forms the basis of a power-management scheme whose aim is to automatically set the optimum performance level of processors supporting dynamic voltage scaling (see Chapter 6).

4.1 Interactive episodes

The beginning of an interactive episode is initiated by the user and is usually signified by a GUI event such as pressing a mouse button or a key on the keyboard. Finding the end of an episode is more difficult since there is no event that automatically gets generated when the computer is done responding. One approach is to assume that user initiated events are CPU bound and to define the end of an episode as the beginning of a relatively long idle section [17]. The length of an interactive episode is thus the elapsed time between a user initiated event (e.g., a mouse click) and the beginning of the next idle period that is longer than a predefined threshold. There are two problems with this approach:



Two typical trace fragments are shown in this figure from the Ghostview benchmark. The first picture shows communication events between tasks after a mouse button was pushed, while the picture on the right corresponds to a complex image being rendered on the screen. The letter R to the right of the pid shows that the task was preempted (the task is ready to execute), a W indicates that it is waiting for an event and gave up time on its own (it is waiting for an event to complete). The arrows indicate the communication flow between the tasks. Task pids in the figure correspond to the following: 757 - X server, 778 - sawmill (window manager), 895 - gnome-terminal, 889 - tasklist_applet, 2088 - Ghostview (gv), 2090 - Ghostscript (gs). Note that Ghostview uses Ghostscript to render pdf and postscript data. In these examples, when a task is waiting for an event, it is blocked in the select or the poll system call.

Figure 4.1: Trace fragments illustrating tasks and communication events

- Episodes that are I/O bound may be terminated prematurely if the wait time exceeds the idle threshold.
- There is a significant latency between the end of an interactive episode and its classification, complicating on-line use of episode information (e.g., for power management).

We developed a more robust episode detection mechanism to alleviate these problems. To find interactive episodes, we keep track of the set of tasks that communicate with each other as a result of a user-initiated GUI event. The start of an interactive episode is signified by the GUI controller (X server in our case) sending a message through a socket to another task. When this happens both the GUI controller and the receiver of the task are added to what we refer to as the task set of the episode. If the members of the task set communicate with non-member tasks, then those tasks are also added. The end of the episode is reached when all the following conditions are met for tasks in the task set:

- No tasks are executing.
- Data written by the tasks have been consumed.
- No task was preempted the last time it ran (i.e., all gave up time on their own by blocking in a system call).

- No tasks are blocked on I/O.

These four conditions allow us to accurately find interactive episodes on both uniprocessor and multiprocessor systems. Intuitively, the end of an episode can only occur when a task is replaced by the idle-thread and if there is no other task (from the task set) executing on any of the other processors. However, if at least one of the tasks is blocked on I/O, then the interactive episode continues until that task is unblocked. Interactive episodes can in some cases contain idle time due to I/O (e.g. Netscape waiting for a page to load over a network, see Section 4.3), however in most cases episodes are CPU bound.

Figure 4.1 illustrates two typical trace fragments from the Ghostview benchmark. In the first case, four processes communicate with the server as the result of a mouse-click event. Note that when a task gives up time, it usually does so in the `poll` or `select` system calls which signifies that the task is ready to process more data. The second trace fragment illustrates the interaction between the X server and a client that is continuously sending data to be displayed. In this case, the Ghostscript renderer is running continuously on CPU 0 and sends the data to the display whenever it has completed rendering a segment. The communication is unidirectional and asynchronous.

An attractive feature of our episode detection technique is that the ends of episodes can be found immediately, without having to wait first for an arbitrary amount of time to elapse. This information can be used on-line by the kernel to make better scheduling and service quality decisions and we demonstrate how it can be used for power management.

4.2 Periodic episodes

While interactive episodes provide feedback about the user-perceived response time of the system, information about periodic episodes can help in determining future utilization of the system. The goal of the mechanism is to infer the period of recurring episodes, and if possible, their processor utilizations. Detecting periodic activity is similar to detecting interactive episodes. However, instead of using communications with the X server as the trigger for starting the episode, we base this decision on the previous execution times of the given task. To detect periodic activity, we keep track of two pieces information for each task:

- Last execution time.

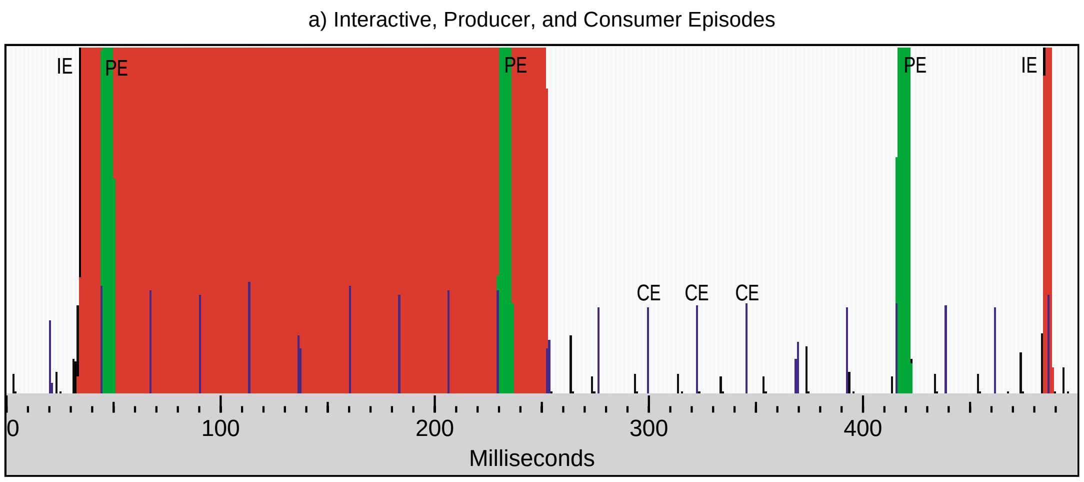
- Length of the *n* last periods.

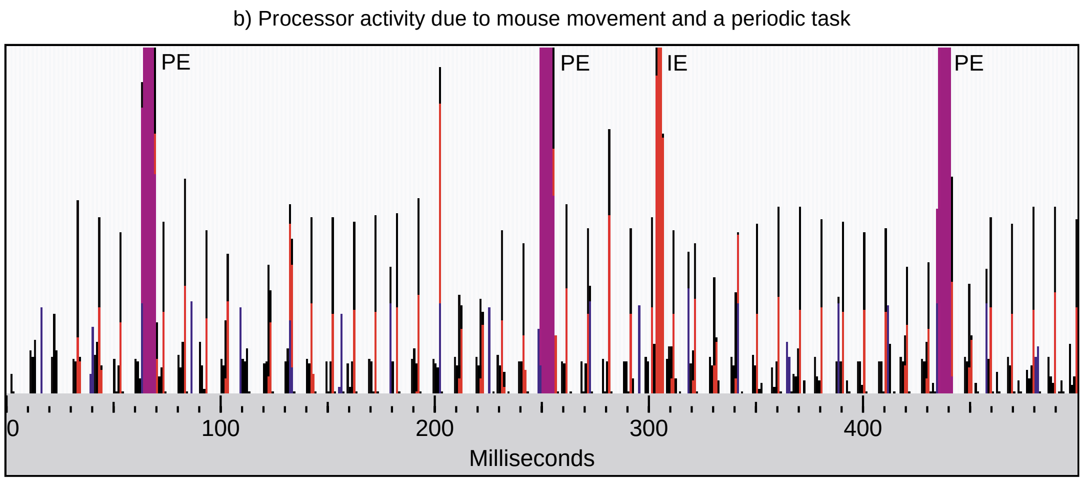
  If a task exhibits only a small amount of variation in period length over the last *n* runs (< 5%), then we treat it as a periodic task. An important subcategory of periodic episodes are the producer and consumer episodes. In addition to these tasks' periodicity, the detection mechanism recognizes their interdependence. A case in point is the Linux esd sound daemon, which wakes up periodically to check for sound playback requests and to send data to the sound card. If esd's playback buffer is not empty, then it sends some of the data to the sound card. If the buffer is close to being empty, then it wakes up and unblocks the music decoders (e.g. MP3 players), which causes them to generate the next few frames of data. In our current studies, we use this information to more accurately gauge the optimum performance level of the processor.

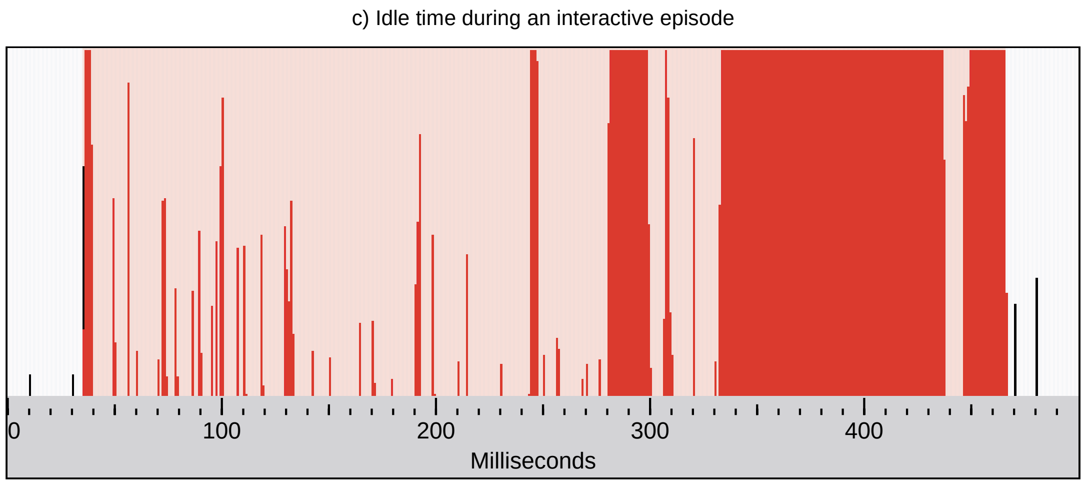
4.3 Examples

Perhaps the easiest way to understand what episode detection accomplishes is to take a look at Figure 4.2. This figure shows three execution traces, where the different types of episodes are highlighted in different colors and an abbreviation of the episode type is shown next to a few key episodes (IE - interactive episode, PE - producer episode, and CE - consumer episode). The episode classification is exactly the same as it would be during run-time, no postprocessing takes place (based on knowledge of the future) to derive the exact begin and end timestamps of the episodes. In the traces, a vertical bar of unit width represents one millisecond of execution. The vertical length of the line corresponds to the utilization of the processor in that quantum. Each line is colored according to the type of episode during the execution quantum (black, if it is not part of any specific episode). In some cases, especially in trace B, a single vertical line may be made up of multiple episodes, in which case the colors are proportional to the execution lengths of each type of episode during the quantum.

Trace A is representative of execution during interactive applications. It includes two significant interactive episodes along with producer and consumer episodes that are triggered by the MP3 player that is executing on the machine. The first thing to note is that the detection mechanism is not confused by overlapping episodes. The MP3 player kicks







The figure shows three execution traces, where the different types of episodes have been highlighted.The first two traces are from the Acrobat reader benchmark and include an MP3 player running in the background, the third trace is from Netscape accessing web servers on the internet (no MP3 playback in background). IE stands for interactive episode, PE for producer episode, and CE for consumer episode.

Figure 4.2: Episodes during execution

in twice (producer episodes) during the first interactive episode, and the classification mechanism accurately attributes the consumed processor time to it. The sound daemon (consumer episodes) wakes up once every 20ms-23ms and runs for about a third of a millisecond. During this time it checks whether there is enough data in the sound card and triggers the producer to decode more data when necessary. This behaviour explains why the producer episode is always preceded immediately by a consumer episode. The checking is accomplished by the sound daemon periodically polling the sound device for data requests using the `select` system call. Note that the only code that gets executed during these short periods is the code corresponding to the `select` system call in the kernel, which checks for activity on the monitored devices. The periodicity of these consumer episodes is determined by how often the kernel schedules processes that are blocked on the `poll` or `select` system calls while waiting for activity. The periodicity can be configured by the caller of the system call but programmers often just rely on the default value in the kernel. When the sound device runs out of data, the `select` call returns—indicating the need to generate more sound data—which in turn wakes up the MP3 player that is blocked in the `write` system call between requests.

This data also illustrates that even a lightweight periodic process can have a significant impact on the user-perceived response time. During the run of the interactive episode, which lasts for 218ms, the MP3 player and the sound daemon use up about 16ms of execution time, causing about an 8% increase of the response time. Section 5.4.4 deals with this issue in more detail.

Trace B shows the bursty activity resulting from the user moving a mouse over the screen, where a document in Acrobat Reader is displayed. In this trace, one sees many short interactive episodes instead of the long ones in Trace A. Just as before, the periodic producer episodes are running, however it is more difficult to visually distinguish the consumer episodes from the interactive episodes. There is only one long interactive episode (at around 300ms), but there are interactive episodes in almost every quantum that has a utilization of about a quarter or more. Short interactive episodes are spaced at about 10ms apart. The distance between these episodes is determined by the X server, which controls the quality of the user experience. Our X server, when it observes rapid mouse movement, increases screen updates to improve the user experience. The interactive episodes are short because the computation required to update the position of the mouse and to redraw affected regions of the screen are very simple. The only heavier-weight interactive episode in this trace lasts for about 3.5ms is a result of a change in the appearance of the cursor when it passes over a special region of the Acrobat Reader application.

Trace C illustrates a long interactive episode with a high percentage of idle time from a run of the Netscape benchmark. The interactive episode starts at around 35ms and ends at 468ms, and it corresponds to a user loading a web page from a server on the internet. The idle time is due to I/O latency while the page is loaded from a remote server. Initially there are only small bursts of activity, mostly dealing with progress updates but as the requested data starts coming in (at around 280ms), the rendering engine kicks in and starts generating output to the screen. Our episode detection mechanism accurately attributes the entire episode as an interactive episode instead of breaking it into smaller disjoint parts. This example also illustrates a shortcoming of our scheme, which we believe is a fundamental problem with kernel level episode detection: for a user, it might be sufficient to wait until the first screen of data is rendered instead of waiting for the entire web page to be ready. However, without modifying the web browser, there is no way of knowing when the window update is done. One could add further hooks into the web browser to accurately signal if the user is really interested in data at the bottom of the page.

4.4 Implementation

Our episode detection mechanisms was purposely designed to be as autonomous from other parts of the kernel as possible. Incorporating these techniques into an existing kernel requires only a small number of hooks, but most importantly it does not require changes to existing scheduling algorithms and policies. This contrasts with the approach taken by other researchers that treat the performance-setting problem as a twist on existing scheduling algorithms [36][24][44]. These schemes usually require perfect knowledge about episode deadlines, which need to be specified either by the programmer or the user of the system.

Most applications under UNIX communicate using sockets, signals, and pipes. In particular, the X server uses sockets to communicate with its clients. We do not track interactions via other methods such as System V IPC and shared memory since our benchmarks do not use them. By tracking the communications between the tasks, we are able to determine which tasks have an effect on interactive performance. Unlike other operating systems (e.g., Windows NT), Linux does not differentiate between threads and processes. Threads are implemented using regular processes and the `clone` system call. We use the name "task" as a synonym for both threads and processes.

The implementation that performs the tracking is as non-invasive as possible. The difficulty was not in the actual implementation but in finding all the parts of the kernel that needed to be tracked. Currently we track communications through the following system calls:

```
kill, pread, pwrite, read, readv, recv, recvfrom,
rcvmsg, send, sendmsg, sendto, write, writev
```

We instrumented each of these system calls to emit a trace of the signals, inodes, and sockets that they are accessing. The socket information is output instead of the inode number, when a socket is accessed through an inode. To be able to match read and write requests through socket pairs, we use the socket's pair (`sock->sk->pair`) on a write and the read socket itself on a read event. Currently we track only communications through UNIX sockets since this is the only socket type that is local to the machine. One could extend this methodology to track communications through other types of sockets if the communicating programs are all local to the machine. However, we have seen no need for this extension so far.

The primary reason for tracking signals is that the thread library (LinuxThreads) uses signals to implement synchronization between threads. By looking at the signal activity we can determine how threads communicate through condition variables, mutexes, and locks. The two functions that needed to be instrumented are `handle_signal` and `send_sig_info`. An alternative to this approach would have been to instrument the thread library; however, our current approach is more generic and has lower overhead.

To determine when tasks are blocked on I/O, we instrumented the `schedule` function to record the reason why it was called. If it is called from a part of the kernel that

A) net/socket.c:

```
int sock_sendmsg(struct socket *sock, struct msghdr *msg, int size)
{
        int err;
        struct scm_cookie scm;

        event_write(sock);
        current->observed_event = EO_SOCK_SENDMSG;

        err = scm_send(sock, msg, &scm);
        if (err >= 0) {
                err = sock->ops->sendmsg(sock, msg, size, &scm);
                scm_destroy(&scm);
        }

        return err;
}
```

B) kernel/sched.c/schedule():

```
        event_thread_swap(current->observed_event, prev, next);

        /*
         * This just switches the register state and the
         * stack.
         */
        switch_to(prev, next, prev);

        __schedule_tail(prev);

        event_task_quantum_begin();
```

Figure 4.3: Examples of modifications in the Linux kernel

is related to I/O (such as the `read` and `write` system calls), then we assume that the task is blocked while waiting for an I/O event to complete. Since there is no predefined way in Linux to find which system call caused a transition to the kernel, we instrumented key system calls to put their id in a field of the executing task's `task_struct`. Once execution gets to the `schedule` function, our code looks at this field and outputs the task's reason for giving up time.

Figure 4.3 illustrates an example of code changes necessary to implement episode monitoring in the Linux kernel. Highlighted lines correspond to modifications made by us, the rest of the code is standard kernel code. The first code example (A) illustrates the changes necessary to a system call that is involved in communications. The `sock_sendmsg` function is responsible for writing data through a socket and can either be invoked through the write or the `sendmsg` system calls. Its main job is to call the appropriate handling function based on socket type. Our changes to the function cause it to call the `event_write` function and update the task's observed_event variable to show that it is currently executing the `sock_sendmsg` code. The `event_write` function is not shown in the figure, however it's main job is to record which task is writing data through the given socket. The last writer of a socket is kept in the socket data structures and is used to reconstruct a map of communicating processes (i.e. task set).

Code snippet B shows a portion of the Linux scheduler, before it switches execution to the next scheduled task. We've instrumented the scheduler to call the `event_thread_swap` function right before execution is transferred to the new task. This function is responsible for keeping track of which tasks are executing on the machine, to record the blocking system call that caused the task to be rescheduled (from the current task's `observed_event` variable) and to keep track of whether the operating system is running useful tasks or is just waiting in the idle loop. The `event_task_quantum` function is invoked when the same task (as the one that was originally rescheduled) starts executing again.

The changes described above form the basis of our monitoring algorithm. In addition to the functions and system calls that are illustrated in this section, we make minor changes to all the system calls that facilitate communications or cause the running process to give up time (e.g. `sleep`, `nanosleep`). Most of the changes to system calls are minor, mostly involved in updating the task's `observed_event` variable. Code that acts on the episode information (such as our performance-setting algorithm) can be hooked in to the kernel through the `event_thread_swap` function, which is called right before tasks are swapped on the machine.

4.5 Conclusion

In this chapter we have described the design and implementation of our episode detection mechanism. Its key advantages are that it works quickly—without incurring latency before detecting the end of an episode and with negligible run-time overhead, accurately—even in the presence of multiprogrammed workloads and overlapping episodes, and its implementation requires only simple and low-overhead changes to the kernel. Our methodology demonstrates that one can derive a lot of information just by looking at communication patterns between the executing tasks and by taking common design patterns into account (e.g. the use of the `select` system call and use of blocking I/O). While in some cases, explicit modifications of the executing user programs could increase the accuracy of our mechanism, we believe that our technique can give most of the benefits without that requirement (Section 6.6 motivates and describes an optional user-level API for conveying episode information).

While our current implementation is tied closely to the Linux kernel and its application environment, we believe that the ideas proposed in this chapter are also applicable to other operating systems. We developed our methodology for Linux by observing common program design and communication patterns. While the specifics may vary from one OS to another, most modern operating systems have abstractions that a similar monitoring environment could be built on (i.e. inter-process communication, multithreading, system calls).

Chapter 5
Thread-level parallelism and interactive performance

Does multiprocessing make sense on the desktop? While our initial investigations in Chapter 3 covered a broad set of benchmarks, the results were inconclusive. There is anecdotal evidence regarding the positive effect of multiprocessing on the “responsiveness” of interactive applications. Intuitively, the premise makes sense: sudden bursts of background activity can be handled concurrently with the foreground task and individual processes can be sped up if they are composed of multiple threads. By applying the episode detection mechanism described in the previous chapter, we investigate whether multiprocessing can indeed affect the user-perceived response time—the time it takes for the computer to respond to user initiated events—of interactive desktop applications.

5.1 Introduction

The primary questions that we deal with is the following: does concurrency translate into improved interactive performance (response time)? The most relevant metric for interactive applications is not the overall throughput but response time: the amount of time it takes for the computer to respond to a user initiated event. We find that desktop applications can incur more than 90% idle time during execution. Running our benchmarks on a dual-processor machine provides a 22% average improvement in application response times. In Section 5.4.4, we investigate the effects on response time of a concurrently running MP3 playback application. Here, the dual-processor machine improves response time by 29% on average. Thus, multiprocessing on the desktop can be a viable means of improving the user experience.

5.2 Metrics and methodology

The principle metrics that we are interested in are idle time (Idle), thread-level parallelism (TLP), and response time (T_R). These metrics (except T_R) are described in detail in Section 3.2.2 and the episode detection algorithm is described in Chapter 4. One signif-

Benchmark	**Version**	**Description**	**Dual processor**			**Uniprocessor**
			TLP_{ie}	**TLP_{run}**	**$Idle_{run}$**	**$Idle_{run}$**
Acroread	4.0	Acrobat PDF file viewer	1.20	1.19	88%	87%
FrameMaker	5.5.6beta	Document editor	1.35	1.33	93%	93%
Ghostview	3.5.8	PostScript and PDF file viewer	1.42	1.39	84%	84%
GIMP	1.1.22	The GNU Image Manipulation Program	1.26	1.24	88%	84%
Netscape	4.7	Web browser	1.34	1.28	90%	89%
Xemacs	21.1 p8	Text editor	1.26	1.21	93%	92%
		Average	**1.31**	**1.27**	**89%**	**88%**

Table 5.1: Benchmark descriptions and characteristics

icant difference from the experiments in Chapter 3 is the version of the Linux kernel: we use the newer 2.3 kernel for the experiments in this and following chapters. Applications running on the new kernel can take advantage of multiprocessing to a significantly higher degree than on the 2.2 kernel.

Idle time is the fraction of measurement time when all of the processors in the system are executing the idle task. Thread-level parallelism, on the other hand, is a measure of how many threads are executing concurrently when the machine is not idle. These two quantities provide a more accurate insight into workload characteristics than machine utilization (MU) alone. While machine utilization can give an accurate picture of the concurrency of the system if idle time is close to zero, it can obscure the presence of concurrent execution in the case of interactive applications, where idle time is high. Response time (T_R) is the length of time between the initiation and completion of an interactive event, which we also refer to as the length of an interactive episode.

5.3 Benchmarks

Table 5.1 gives a short description of our benchmarks and summarizes their high-level characteristics. The data presented in this paper are averages of seven benchmark runs in each configuration. All benchmarks were run by a live user. While we aimed to repeat each run as accurately as possible, there are slight variations between the runs. All the significant events (e.g., mouse clicks, text entry) were performed in the same order during each benchmark run. However, the exact path of mouse movement (and therefore the interactive episodes corresponding to them) and the amount of time between events varies from one run to the other.

All our applications show significant amounts of TLP and a high fraction of idle time. It is likely that the idle time of the application executing on an actual user's desktop would be higher since although we interacted manually, we made no efforts to consume all the information presented by the program. The overall TLP of our applications are similar to what we measured during the interactive episodes. However, in all cases the TLP in interactive episodes was higher than the average for the entire run of the benchmarks.

5.4 Response time results

One way of quantifying an application's performance is to measure the time it takes to complete a run of the benchmark. This approach works for throughput-oriented benchmarks, however it runs into difficulties when one tries to measure interactive applications. Nonetheless, benchmarks such as Sysmark 98 [46] and Winstone 99 [49] attempt to quantify the performance of interactive applications by turning them into throughput-oriented benchmarks. They accomplish this by using a software driver that clicks through these applications as quickly as possible and by measuring the length of the end-to-end execution.

The problem with this approach is that in interactive applications not all parts of the program's execution are equally important. The relevant metric is improvement in response time (the time it takes to respond to a user-initiated event) of critical episodes [11]. This time has also been called "wait time," to refer to the fact that during these periods the user is actively waiting for the computer to complete the task.

This section presents measured response times from uni- and dual-processor machines. The first subsection compares the performance of individual interactive episodes while the second analyzes and compares aggregate statistics from full benchmark runs. Section 5.4.3 further analyzes these results in the context of user-perceptible improvement. Finally, Section 5.4.4 repeats the initial experiments with an added active background process (an MP3 player).

Benchmark	Average response-time improvement of mouse-click events	Selected episodes					
		Episode description	Response time (T_R) improvement	Dual processor		Uniprocessor	
				TLP$_{ie}$	T_R (sec)	Measured T_R (sec)	Predicted T_R (sec)
Acroread	15%	Displaying successive pages of a pdf file.	14%	1.21	0.119	0.138	0.144
			17%	1.21	0.125	0.150	0.151
			12%	1.15	0.231	0.261	0.264
FrameMaker	22%	Visually manipulating a FrameMaker document.	30%	1.43	0.296	0.425	0.422
			20%	1.29	0.040	0.050	0.051
			25%	1.41	0.022	0.029	0.031
			27%	1.38	0.021	0.029	0.029
Ghostview	34%	Displaying successive pages of a pdf file.	36%	1.51	0.223	0.346	0.336
			19%	1.29	0.455	0.562	0.586
			36%	1.53	0.225	0.352	0.345
			30%	1.47	0.403	0.578	0.593
			32%	1.52	0.331	0.484	0.502
GIMP	19%	Pixelize	18%	1.37	0.456	0.553	0.623
		Motion blur	8%	1.15	1.206	1.312	1.390
		Sharpen	20%	1.33	0.408	0.511	0.541
		Laplace edge-detect	15%	1.19	0.982	1.156	1.164
		Undo	22%	1.38	0.118	0.152	0.164
Netscape	21%	Displaying simple HTML pages from a machine-local web server.	24%	1.34	0.252	0.331	0.338
			25%	1.42	0.096	0.127	0.137
			20%	1.31	0.064	0.079	0.084
			28%	1.44	0.085	0.118	0.123
Average	**22%**						

Table 5.2: Response time on a dual-processor and a uniprocessor machine

5.4.1 Individual episodes

We have noted that while the runs of our benchmarks are similar to each other, they are not completely identical. This poses problems when we attempt to compare two different runs to each other. We cannot just compare the average lengths of interactive episodes from one run to the other, since the set of episodes in each run could be slightly different. We overcame this problem by comparing only individual episodes that occur after the same mouse click in each trace. These episodes are more repeatable than those caused by other events, such as focus changes, and tend to have longer response times.

To correlate mouse-click events with interactive episodes we modified the X server to write an entry into the trace every time a mouse button is pressed. The postprocessor can then correlate interactive episodes that occur after a marker. Table 5.2 summarizes the results of the response time measurements for these episodes. The first column includes the overall response time improvement for the mouse-click episodes in the benchmark, while the rest of the columns show the collected data and the response time improvement for a few selected episodes. Note that, since our Xemacs workload was driven solely by keyboard interactions, we were unable to compute response time improvement for it.

The results indicate that on our benchmarks response time improved on a dual-processor machine by an average of 22% (36% in the best case, 8% in the worst). The average TLP for the episodes is 1.31, higher than the average for the complete benchmarks, which is 1.27. Idle time during the interactive episodes in all cases was zero or very close to it (a few tenths of a percent).

To check our results we used the dual-processor numbers to estimate the uniprocessor run-time (Equation 5) and then checked them against actual measured values. DP refers to the response time (T_R) and idle time (T_{Idle}) on a dual processor, while UP refers to the same measurements on a uniprocessor machine.

$$T_{R(UP)} = (T_{R(DP)} - T_{Idle(DP)})TLP + T_{Idle(DP)} \qquad \text{(EQ 5)}$$

The equation scales the non-idle portions of the episode by the measured TLP and assumes that no scaling occurs on the idle portions. This simple model predicts the uniprocessor episode lengths to within 4% on average (one run had an error of 11% and for all others, error was under 7%). Given that we made no special provisions to reduce experimental variations (e.g., by turning off background daemons) and that all traces were driven by a real user (instead of an automated script), we think that the error is within a reasonable margin.

Most Linux applications are not threaded; concurrency emerges from simultaneously running multiple processes. The only applications from our benchmarks that actually used threads (through the LinuxThreads API) were Netscape and GIMP. These applications derived some TLP by running intra-application threads concurrently. However, most of the TLP was achieved by running the application thread concurrently with the user interface threads: mostly with the X server but also with the other GUI tasks (such as the window manager, desktop applets, etc.). This contrasts with our experience under Windows NT, where application threads ran concurrently primarily with threads from the System process, which includes device drivers and other operating system threads (Chapter 3).

5.4.2 All interactive episodes

In the previous section we looked at select episodes that come after mouse clicks to figure out the response time improvement. These episodes usually represent the heavyweight episodes during the benchmark runs and, while these episodes usually make up the largest percentage of time during the run, the number of short episodes dominates.

Table 5.3 shows the episode length distribution of our benchmarks. Due to the large variance of episode lengths, we separated the results into four categories. For each category, the number of episodes that fall into it are given (percentage of episodes) along with the total amount of time spent (as a percentage of total time in all interactive episodes). While the majority of the episodes are very short and are in the few tenths of a millisecond range, only a small portion of the time is spent in episodes corresponding to them. This makes sense given the orders-of-magnitude variance in episode lengths. Examples of the short episodes include:

- Moving the mouse and updating its position.
- Updating the appearance of the cursor.
- Handling window focus changes.
- Handling keyboard events.

Towards the right hand side of the table is the data corresponding to the heavyweight episodes that were the subject of our investigations in the previous section. Since in the majority of our benchmarks most of the time is spent in executing these kinds of episodes and most of them fall above the perception threshold of the user, these are of primary importance for speeding up.

Table 5.4 gives the TLP and average length of interactive episodes. The most significant observation is that in all cases TLP is higher in the interactive episodes than the

Benchmark	[0ms, 1ms)		[1ms, 10ms)		[10ms, 100ms)		[100ms, inf)	
	% of episodes	% of time	% of episodes	% of time	% of episodes	% of time	% of episodes	% of time
Acroread	92.69%	5.75%	4.11%	3.52%	1.13%	11.89%	2.08%	78.85%
FrameMaker	72.10%	2.87%	17.60%	6.98%	8.58%	42.11%	1.72%	48.04%
Ghostview	89.87%	2.24%	6.73%	2.22%	0.76%	6.7%	2.64%	88.85%
GIMP	87.93%	2.7%	10.19%	5.89%	0.32%	0.64%	1.57%	90.77%
Netscape	89.98%	3.56%	8.62%	13.88%	0.98%	31.43%	0.42%	51.13%
Xemacs	65.01%	4.78%	34.36%	86.01%	0.63%	9.21%	0%	0%

Table 5.3: Episode distribution (dual processor)

Benchmark	[0ms, 1ms)		[1ms, 10ms)		[10ms, 100ms)		[100ms, inf)	
	TLP_{ie}	avg. length	TLP_{ie}	avg. length	TLP_{ie}	avg. length	TLP_{ie}	avg. length
Acroread	1.38	0.25	1.19	3.47	1.20	42.76	1.18	153.66
FrameMaker	1.38	0.30	1.20	2.94	1.36	36.42	1.37	207.72
Ghostview	1.30	0.23	1.18	3.10	1.33	83.39	1.43	315.85
GIMP	1.43	0.17	1.22	3.28	1.35	11.49	1.26	328.83
Netscape	1.70	0.07	1.16	2.73	1.41	55.59	1.33	210.60
Xemacs	1.87	0.07	1.24	2.52	1.14	14.68	N?A	N/A

Table 5.4: TLP and episode length distribution (dual processor)

average for the entire run of the benchmark (see Table 5.1). This matches with our observation in the previous section. Moreover, in all cases the interactive episodes appear to be very CPU bound, with zero or close zero idle time.

Short episodes (less than one millisecond long) tend to have the highest TLP. This should not be surprising since these episodes usually perform an update of a few GUI objects on the screen, which requires the tight interaction of both the X server and the client. The TLP in the most used category varies from one benchmark to the other. It is never smaller than the overall average for the program but in some cases it is smaller than the average for the interactive episodes.

While most time is spent executing episodes that fall in the tenth of a second to over a second range, some last for only a fraction of a millisecond. With episodes that are

Benchmark	100ms threshold				50ms threshold			
	Dual processor		Uniprocessor		Dual processor		Uniprocessor	
	% time	# episode	% time	# episode	% time	# episode	% time	# episode
Acroread	79%	8	85%	9	89%	9	90%	9
FrameMaker	48%	2	49%	2	69%	5	75%	6
Ghostview	89%	12	95%	15	96%	15	96%	15
GIMP	91%	9	92%	9	91%	9	92%	9
Netscape	51%	4	63%	7	72%	10	73%	10
Xemacs	0%	0	0%	0	0%	0	0%	0

Table 5.5: Episodes above the perception threshold

so short, the question comes up whether there is any perceptible improvement in responsiveness using two processors.

5.4.3 The perception threshold

We have shown that TLP can be exploited successfully to reduce the response time of interactive applications. Once the response time reaches a certain threshold, the user is not able to detect any further improvement. What exactly that threshold is depends on the event type and can vary from one user to another. While its actual value is hard to quantify, the perception threshold sets an upper bound for the required performance.

Determining the exact length of an interactive episode can also be problematic. Does an episode begin when the user clicks the mouse button, or when that event is delivered by the X server? We have taken the position that interactive episodes begin when the event is delivered, since the X server may need to wait for additional events—such as extra mouse clicks to distinguish a double-click from single-click or for a button release—before delivering the event. Our measurements show that the delay between the hardware event and event dispatch can vary from a few tenths to hundreds of milliseconds. When considering an appropriate perception threshold for a user, this extra delay may need to be accounted for. We do not address this problem in this paper.

Table 5.5 shows the number and fraction of time spent in episodes that are above the perception threshold. The fraction of time is expressed as the percentage of time in all interactive episodes. Data is computed for two threshold values, which were selected based on data from [6]. Given either threshold, most of the time is spent executing episodes that fall above the perception threshold. Based on this data, FrameMaker, Netscape, and Xemacs are the most responsive applications. This correlates well with our experience; all three of these applications seemed to be very responsive and we could not experience any qualitative difference between the dual-processor and uniprocessor runs of these applications. We must note, however, that while interactive episodes in Netscape were under the perception threshold when accessing a web server on the local machine, accessing servers on the Internet would certainly show more episodes in the perceptible range due to network latency. However, network latency is not something that multiprocessing in the client can reduce.

While exploiting TLP reduces the average length of the interactive episodes, it only causes a shift of an interactive episode from above to below the perception threshold, if its length on a uniprocessor does not greatly exceed the threshold. None of the benchmarks had a significant shift in the number of episodes when the perception threshold is set to 50ms, and only a few of our benchmarks' episodes moved below the cutoff at the 100ms threshold (Acroread, GhostView, and Netscape).

5.4.4 Effects of background activity

To round out our investigations, we wanted to know what happens when a background process is executing along with the interactive application. To gain some insight into such workloads, we repeated our experiments with an MP3 player running in the background. We used a very simple MP3 player called mpg123 (version 0.59r) along with the esd sound daemon. The player lacks a graphical interface and only does music playback—no visual effects. This application is light-weight and exhibits very little concurrency. We measured 1.02 TLP and 95% idle time when running music playback by itself.

The results are presented in Table 5.6. The response times are averages over all mouse-click events, as in Table 5.2. The performance improvements due to two processors is more significant than in our previous measurements. The average improvement is 29%, in contrast to 22% without the background task. On a dual processor the work required for MP3 playback is mostly absorbed by the extra processor. However, on a uni-

Benchmark	TLP_{ie}	TLP_{run}	Response time improvement	Response time increase due to MP3 playback	
				Dual processor	Uniprocessor
Acroread	1.25	1.19	23%	4%	15%
FrameMaker	1.40	1.20	29%	1%	13%
GhostView	1.46	1.34	38%	4%	10%
GIMP	1.32	1.23	23%	4%	14%
Netscape	1.39	1.24	31%	5%	16%
Xemacs	1.35	1.18	N/A	N/A	N/A
Average	**1.36**	**1.23**	**29%**	**4%**	**14%**

Table 5.6: Response time improvement on dual- and uni-processor machines with MP3 playback

processor the extra work cannot be off-loaded and must be performed during the critical path, thus extending the lengths of the interactive episodes. Compared to our previous results, the dual-processor episode lengths are increased by an average of 4%, while the uniprocessor episode lengths are increased by an average of 14%.

The average TLP within the interactive episodes increases to 1.36 while the average for the entire benchmark run decreases to 1.23. The trend is the same as in our previous measurements without MP3 playback. However, in this case the difference between TLP_{ie} and TLP_{run} is significantly greater (the average TLP_{ie} is 1.31 and TLP_{run} is 1.27 when there is no MP3 playback in the background). The reason for the greater difference is that MP3 playback is periodic and has no inherent concurrency (it is not threaded, just a single task), which affects TLP in two ways:

- It reduces idle time and the new non-idle portions have a TLP of one.
- On existing non-idle periods, it increases TLP.

Since interactive episodes have very little idle time, TLP goes up due to the concurrently running MP3 playback process. On the non-interactive portions, the background application reduces idle time and replaces the idle thread with a single running application. Since all of our benchmarks are dominated by idle time, the TLP_{run} of the applications is the same or lower with MP3 playback in the background than without.

5.5 Conclusions

The fundamental question that we wanted to answer was whether it is beneficial for a desktop user to use a multiprocessor machine for everyday tasks. We have shown that existing Linux workloads exhibit thread-level parallelism, which translates into improvement of the user-perceived response time of the applications. Using two processors instead of one is a straightforward way to reduce execution length in the critical path in our benchmarks by 8% to 36% (22% on average). Moreover, these improvements represent 16% to 72% of the maximum reduction achievable on a dual-processor machine (50%). Using two processors can thus be an effective and efficient way of improving interactive performance.

The average response time improvement on a dual-processor machine increases to 29% with an MP3 player executing in the background. Although the extra processor eliminates most of the overhead of a background task, it does not absorb it all. The average length of an interactive episode increased by 4% due to audio playback (vs. 14% on the uniprocessor). This result is consistent with the level of TLP we found in interactive applications: we should expect the response time to remain unchanged only if the second processor is completely unused by the foreground application.

For most of our applications using more than two processors is not likely to yield great improvements. This conclusion is supported by our previous experience on a quad-processor machine, where the only workloads that had a TLP of 2 or more were hand-parallelized or were batch jobs (see Chapter 3). Our current results show that most workloads have TLP under 1.4, which implies that increasing the number of processors would only be beneficial in less than 40% percent of the time (i.e., the proportion of the episode where TLP is 1 is not reduced). Even by making the optimistic assumption that in all these episodes four threads could run concurrently 40% of the time, we can only expect an overall speedup of 20% on a quad-processor over a dual-processor machine.

In our experience most of the concurrency was achieved by overlaying the execution of the GUI controller (X server) with an application task that is communicating with the GUI. We have noted that the utilization of processors is usually unbalanced. This leaves some room for software designers to repartition interfaces in order to more efficiently utilize the hardware. In particular, a higher level API for rendering images in the X server could improve the balance between the server and the clients. A more general approach to balancing would be to run the CPUs in the system at different levels of performance depending on the particular workload. This optimization would increase TLP and decrease energy consumption.

Historically, uniprocessor performance has doubled every 18 months. Given a TLP of 1.5, this means that the lead time of a dual processor over an equivalent-performing uniprocessor is about three quarters of a year. Given that most interactive episodes in our measurements were shorter than 500ms, in about three years these episodes will fall under the 100ms perception threshold on a dual-processor machine and in about four years it will be sufficiently fast on a uniprocessor. On the other hand, the software four years from now will likely increase its processing requirements.

In our opinion, our results indicate that the Linux kernel and associated software have come of age as an SMP platform. While thread-level parallelism can certainly be further improved by removing uses of the global kernel lock, we believe that the emphasis should now be on application writers to refactor their programs with multithreading in mind. Multiprocessing would become even more compelling if TLP could be increased above 1.5.

Chapter 6
Automatic performance-setting for dynamic voltage scaling

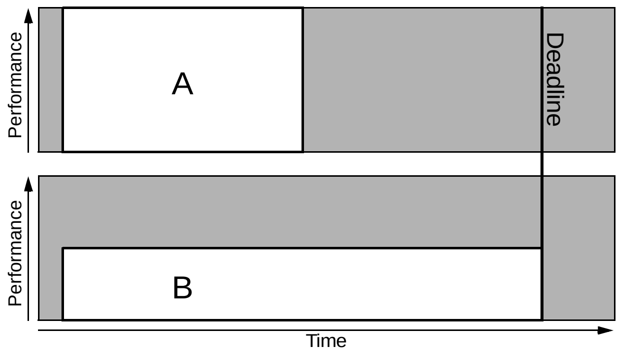
6.1 Introduction

The performance of microprocessors has been improving at an exponential rate and this trend is likely to continue for several years to come. However, increased performance does not come for free. One of the most important consequences of higher performance has been a dramatic increase in power consumption. While an Intel 386 processor consumed about 2 Watts of energy, a Pentium 4 can use as much as 55 Watts. In a mobile environment, batteries have not kept pace with the increased energy requirements, which means that either application performance or battery time suffers. However, even in environments where energy storage is not an issue, energy cost and heat management may become problems [12].

There is still a need to continue to improve processor performance, since not all applications are “fast enough,” but an increasing number are. A way to bridge the gap is to allow the processor to run at different performance levels depending on the application’s requirements. Some processors, such as the Intel XScale [23] and Transmeta Crusoe [25] allow the frequency of the processor to be reduced with proportional reduction in voltage. Slowing down frequency without voltage scaling is not sufficient, since the power savings is offset by an equal increase in execution time, thus yielding no reduction in the total amount of energy consumed. However, since energy is proportional to the square of the voltage, reducing the operating voltage can yield significant energy savings [40].

The central issue with processors whose performance can be changed to save energy is how the right level of performance can be obtained. The goal is to reduce the performance of the processor without causing an application to miss its deadlines (see Figure 6.1). Completing a task before its deadline with consequent idling is less energy efficient than running the task more slowly to begin with and meeting its deadline exactly.

Our aim is to design an algorithm that balances energy savings with the following requirements:



This figure shows two different runs of the same workload. In A, the workload runs at full speed and finishes well in advance of its deadline. In B, the execution of the workload is stretched to its deadline, which allows for energy savings on processors that implement voltage scaling.

Figure 6.1: Performance scaling

- No modification of user programs.
- Works with irregular and multiprogrammed workloads.
- Ensures that the user-perceived performance does not suffer.

Previous interval-based approaches to automated performance setting did not fully achieve the goals outlined in the last two points. These approaches focus on the ratio of idle- to busy-time as the indicator of the right performance setting [40][48][17][18][38]. While the results looked promising for regular workloads (such as audio playback where processor utilization is periodic), the proposed schemes do not work well for interactive or irregular applications.

The aforementioned papers point out that looking at idle time alone as the indicator of the right performance level is not sufficient. In their future works section, Weiser et al. propose an alternative approach, where jobs are classified into background, periodic, and foreground classes [48]. They suggest that the added semantic information could be used to improve the scheduling algorithms. Govil et al., in their future works section, propose a similar solution, where process type along with information specified by the processes (e.g. deadline) could be used for performance setting [17]. Our approach follows along the lines of these earlier works. However, we derive deadline and classification information automatically from the OS kernel, by examining the communication patterns between the executing tasks. This information is used to guide performance-setting decisions on a per-task basis and also to dynamically evaluate the impact of past decisions on user-perceived performance. We can classify execution episodes into one of the following categories: interactive and periodic (with producer and consumer subcategories). These classifications can be used to derive deadlines for the execution episodes. For example, for an interactive episode, the deadline is the perception threshold, which we assume to be between 50ms and 100ms. The deadline for a producer is the point at which the consumer actually needs the produced data.

We focus primarily on interactive applications, since we believe that this is one of the most difficult but also the most important class of applications for performance scaling. We also consider the effects of a concurrently running background application (an MP3 player) and a variety of assumptions about the power and performance models on the performance-setting strategy (Section 6.4).

6.2 Performance prediction

Our prediction mechanism operates on a per-task basis and uses different algorithms for interactive and periodic episodes. In both cases, the predictor computes the performance factor, which is the ratio of the desired execution speed and the processor's maximum speed.

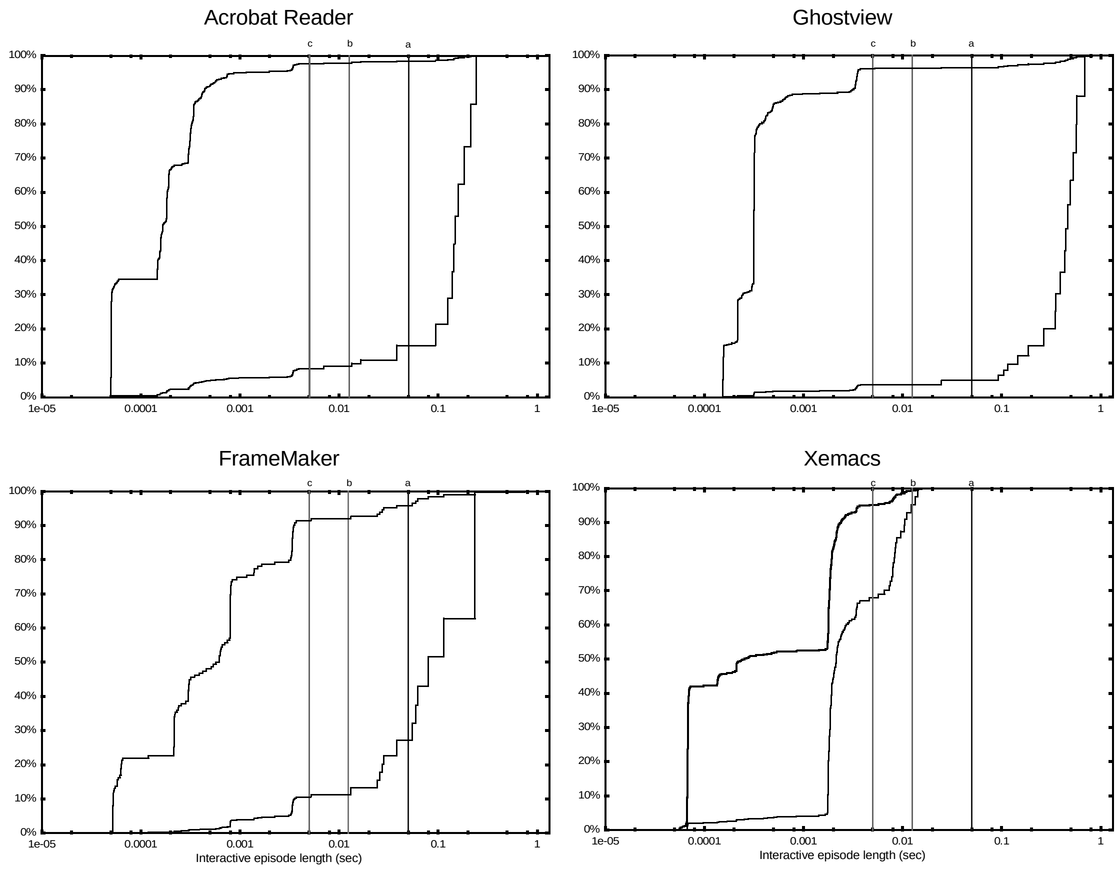
6.2.1 Interactive episodes

It is difficult to come up with a good prediction strategy for the optimum performance level of interactive episodes, since interactive episodes are completely dependent on the user, not on some activity within the computer. There is no easily predictable pattern of recurrence, and the lengths of interactive episodes can have orders of magnitude of difference. Our detection scheme allows us to differentiate between different types of episodes (i.e. interactive, producer, consumer) but cannot distinguish between different instances of the same episode in the same task (e.g. when the same button is pushed in the GUI as before).

One aid to prediction would be the ability to distinguish between different types of interactive episodes. However, this would require the kernel to have knowledge about the location in the user program that initiated the given interactive episode. While not impossible, distinguishing the real call-sites from the kernel is difficult to do. A simple comparison based on the user-mode program counter (PC) is not sufficient, since programs usually go through at least one level of indirection (through libC) when calling a system call, and thus all instances of a program's calls to a given system call would have the same user-mode PC. Moreover, since interactive episodes are usually generated as a result of GUI interaction, the necessary number of indirection levels is probably higher due to the use of GUI libraries (e.g. gtk, Xlib). To find the PC value that really distinguishes one interactive episode from another, one would have to chase pointers through multiple levels, where the actual number of levels depends on the environment (stack layout, libraries, etc.).

Instead of basing a predictor on the ability to distinguish between interactive episodes, we looked for a simpler solution that only relies on episode type for prediction.

Figure 6.2 shows the cumulative distribution of interactive episode lengths for four interactive benchmarks. In each graph, there are two cumulative distributions: the one on the left shows the cumulative number and the one on the right shows the cumulative time spent in interactive episodes of a given length or shorter. To account for the large variation of interactive episode lengths, the time axis is logarithmic. Three vertical lines (a, b, and c from right to left) delineate the perception threshold (50ms), the point under which all episodes finish under the perception threshold at 1/4th of peak performance (12.5ms), and 1/10th peak performance (5ms). These values were selected because current processors that are capable of performance and voltage scaling have a minimum performance of about 1/4th peak performance, and future processors could possibly extend the rage of performances to 1/10th of peak value. These graphs show that while most episodes are very short, the vast majority of time is actually spent in a small fraction that correspond to the long episodes. For example, in Ghostview, 92% of the time is spent in 4% of the episodes. The Xemacs benchmark is an example of an application where one could run almost all of its interactive episodes in the lowest performance level without ever exceeding the perception threshold.



Left line shows the cumulative number, right line the cumulative percentage of time spent in interactive episodes whose lengths are less than or equal to the time specified on the x axis. The x axis is drawn using a logarithmic scale. Vertical lines from right to left: a) 50ms, b) 12.5ms and c) 5ms.

Figure 6.2: Cumulative interactive episode length distribution

The cumulative episode length distribution graphs imply that a predictor that predicts that an interactive episode only needs the minimum available processor performance would be right more than 90% of the time. However, since these episodes tend to make up only a small percentage of total time—and consequently have a small contribution to energy use—it is more important to focus on accurately predicting the performance level of the relatively long episodes.

Our performance-factor predictor for interactive episodes works by starting off with an initial performance factor, set to the minimum performance factor of the processor, and then successively refining its value. The algorithm uses the following three steps:

- Start running the episode at the predicted performance factor.

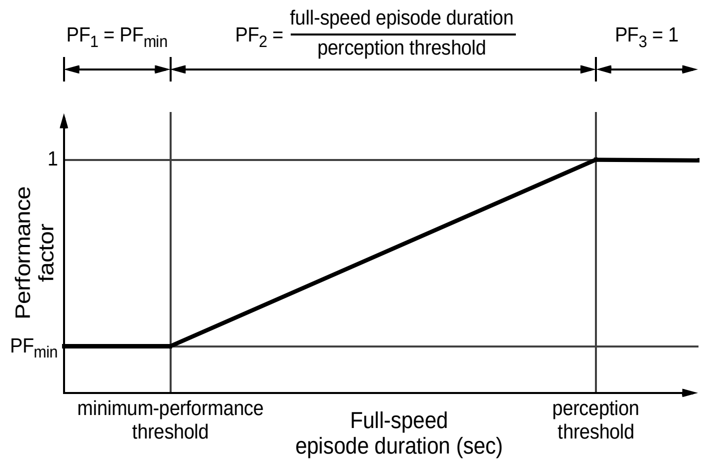
- At the end of the episode, compute the duration that corresponds to executing at full performance. Use this information to compute the optimal performance factor for the episode.
- Use the weighted average of optimum performance factors (*PF*) as a prediction for future performance factors.

The main observation that we use in our predictor is that it is straight forward to compute what the optimum performance level should have been once an interactive episode is over. During the execution of the episode, the performance-setting of the processor might be changed by external events (e.g. periodic episodes start executing), so the algorithm must keep track of the observed performance factors (pf_i) during the episode's execution. At the end of the episode, this information can be used to estimate how long the episode would have been at full performance (see Equation 6).

$$T_{FullSpeed} = \sum_{i=1}^{n} pf_i(t_i - idle_i) + idle_i \qquad \text{(EQ 6)}$$

This equation computes the full speed execution time for an interactive episode given n different observed performance levels during the episode. The variable t_i specifies the length of execution during an interactive episode, and $idle_i$ the amount of idle time during execution at the *i*-th performance level (pf_i).

Figure 6.3 illustrates how the optimum performance factor can be computed at the end of the interactive episode based on the full-speed episode duration. There are two significant inflection points in this graph: the perception threshold, and the minimum-performance threshold. The minimum-performance threshold specifies the full speed episode duration that could be slowed down to the processor's minimum performance level and still finish under the perception threshold. If the perception threshold is set to 50ms and the processor's minimum performance is 1/4th of peak, then this value is 12.5ms. Episodes that are shorter than the minimum-performance threshold can be run at minimum performance (PF_1). Episodes that are longer than the perception threshold need to run at full performance (PF_3). Episodes that fall between these two thresholds can be slowed down to be exactly as long as the perception threshold (PF_2). If there is a significant amount of idle time during the episode, then the idle time needs to be subtracted from both the numerator and the denominator when computing PF_2.



The above formulas give the level of performance that minimizes energy use for a given episode length (performance factor). The minimum-performance threshold specifies the episode length below which episodes finish under the perception threshold even in the slowest performance setting. When an episode is longer than the minimum-performance threshold but shorter than the perception threshold, it can be slowed down to be exactly as long as the perception threshold. In this model, perceptible episodes need full performance.

Figure 6.3: Computing the performance factor for interactive episodes

The prediction for the next interactive episode of a given task is simply the weighted average of the optimum performance factors of past interactive episodes.

$$PF_{prediction} = \frac{\sum_{j=1}^{k} PF_j T_j}{\sum_{j=1}^{k} T_j} \qquad \text{(EQ 7)}$$

Equation 7 shows the computation for the predicted performance factor based on the optimum performance factors (PF_j) for k past interactive episodes. T_j refers to the estimated full-speed time of an interactive episode. The size of k can be varied to eliminate saturation and to allow temporal variations of episode lengths to affect the predictor (see Section 6.2.4).

Our predictor does not look for recurring patterns of episode lengths, it simply sets the performance to the level that is the expected value based on the weighted average of the lengths of past episodes. Since there can be orders of magnitude difference between the lengths of interactive episodes (see Figure 6.2), this strategy means that the predicted performance factor for short episodes will almost certainly be higher than necessary. This effect is mitigated by the observation that short episodes have only a small effect on energy consumption.

To recover from prediction errors we set an episode-duration threshold, after which if the episode is still executing, the performance level is raised to full speed. We refer to this threshold as the PanicThreshold. While the PanicThreshold can ensure that interactive performance does not degrade below a certain level, the goal of the predictor is to set the right performance level at the beginning of the episode, without the need to transition to a higher performance setting later on.

The setting for the PanicThreshold reflects the user's tolerance for worst-case performance degradation and determines how speculative the performance factor predictor can be. If the user has no tolerance for possible performance degradation, there is no opportunity for speculation and consequently energy savings. In this case one would be forced to be conservative and always run at full performance to avoid misprediction errors that might extend the episode beyond the perception threshold.

Equation 8 shows the formula for the PanicThreshold for a given performance factor (*PF*) and perception threshold. The performance factor refers to the predicted performance factor at the beginning of the interactive episode.

$$PanicThreshold = PerceptionThreshold(1 + PF) \qquad \text{(EQ 8)}$$

This formula makes the assumption that the user allows more performance degradation for episodes that are short, than for long ones. For example, if the performance factor is set to 0.25 (and assuming 50ms perception threshold), then the length of an episode that would have taken 50ms at full speed would be extended to 97ms. However, given the same parameters, an episode that would have been 200ms at full speed would only be stretched to 247ms. The use of the performance factor in the formula allows more time for speculation when the initial performance level is high. This formula was derived based on information in [5], however, due to the lack of an actual computer that supports dynamic voltage scaling, we have not been able to personally evaluate its impact on the user-perceived performance. We also tried a simple formula, where the panic threshold was set to be the perception threshold regardless of the performance factor, and found the difference in our energy results to be negligible.

6.2.2 The perception threshold

In this paper we use a range of perception thresholds during some of our experiments. Our motivations for this are twofold:

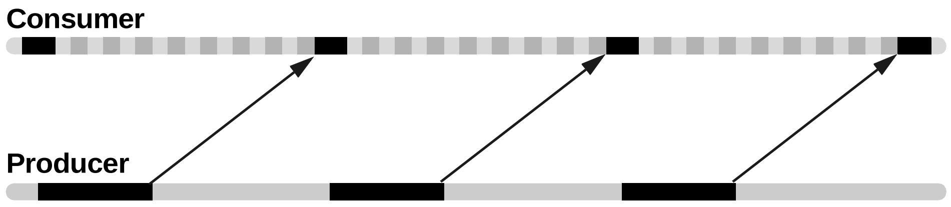
- The higher perception thresholds allow us to estimate the energy and interactive characteristics on a future, higher-performance processor. The 100ms threshold on today's processor roughly corresponds to the 50ms threshold on a processor with twice the performance.
- The perception threshold varies by individual and task and may be used as a user settable indicator for his preference for high-performance or energy savings.

Literature about human-computer interaction [37][5] indicates that 20-30 frames per second are sufficient for the human visual system to perceive the images as a continuous stream. This suggests that the perception threshold is around 50ms. Human subject tests in [5] show that perceptual causality—when two events are perceived to be fused together—ends around 100ms, and for some test subjects quality degradation begins at around 50ms.

Other experiments have shown that for simple operations, such as dragging an object through the screen, as few as 5 updates per second are sufficient to maintain an interactive feel (200ms perception threshold). For non-continuous operations, as much as 1-2 second delays are acceptable [37]. However, when human motor operations form a feedback loop with visual activity, then it is more important to have a faster response time.

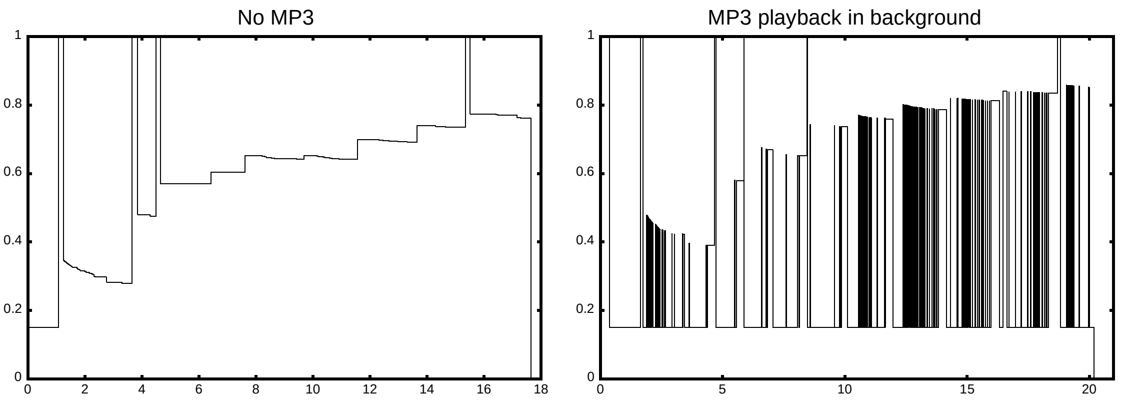
6.2.3 Incorporating periodic episodes

The optimum performance factor for periodic activity can be computed easily by either stretching the periodic episode's execution to the beginning of the next episode or to the beginning of the associated consumer episode (Figure 6.4). Since periodic (such as video or sound playback) applications sometimes adjust the quality of playback based on processor performance, it is important to switch to full performance when a periodic appli-



The figure shows communications between a producer and a consumer process. The processor can be slowed down to stretch the producer episode to the beginning of the consumer episode.

Figure 6.4: Producer and consumer episodes



Two runs of the Acrobat Reader benchmarks are shown side by side with and without MP3 playback during the run. Perception threshold was set to 200ms, and data was generated using our simplest strategy: the Basic predictor and XSB model without quantization. More sophisticated models have significantly fewer performance-level transitions when MP3 is executing in the background. Spikes to full performance represent instances when the PanicThreshold was reached. Note that the graphs do not show transitions to sleep mode.

Figure 6.5: Performance-setting using the Basic predictor (XSB model)

cation starts executing, so that it has a chance to adapt to the highest performance level. Our assumption is that the user's emphasis is on service quality over energy savings. Others have addressed the trade-off where service quality can be reduced to save energy [15].

An important consideration is to find the performance factor when interactive episodes are present in addition to the periodic activity. Our strategy is very simple:

- When there is no interactive episode executing on the processor, we set the performance factor to the one computed for the periodic activity.
- At the beginning of an interactive episode we switch to the performance factor that was predicted for the task's interactive episodes, if it is higher than the periodic performance factor.

Figure 6.5 illustrates this strategy during two runs of the Acrobat Reader benchmark. When there is no periodic activity, performance is determined only by the prediction for interactive episodes. However, when periodic activity is present, the algorithm switches between the two performance levels, causing significantly more performance transitions. The spikes that transition the processor to full performance are triggered by interactive episodes whose lengths exceed the PanicThreshold. Aside from the initial start at full performance when MP3 is executing in the background, there is only one extra transition due to reaching the PanicThreshold on the second figure.

Periodic episodes have the effect of extending the run-times of interactive episodes (see Chapter 3), which means that the interactive performance-factor predictor should be updated when periodic activity starts. Instead of making the prediction formula more complicated, our approach allows the performance predictor to quickly adapt to the presence of the periodic activity. The next section describes how this is done exactly.

6.2.4 Implementation details of the Basic predictor

The main features of the Basic predictor are summarized below:

- Interactive episode performance level prediction based on optimum performance factors of past episodes.
- PanicThreshold bounds worst-case performance.
- Periodic performance level computed by observing periodic episodes and their communication patterns.
- Switches between periodic and interactive performance factors depending on which episode is executing.

An attractive feature of the predictor is that it requires very little state. We use two floating-point variables per task: one keeps track of the sum of episode length weighted performance factors (*totalPF*) and the other keeps track of the total time spent in episodes (*totalTime*). In both cases time is the estimated full-speed execution time of the episode. The performance factor prediction is thus *totalPF* divided by *totalTime*.

One problem with an averaging based predictor is that if the execution time is long, then temporal variation may not influence the predictor for a very long time. An easy way to alleviate this problem is to periodically rescale the variables by dividing them both by the same amount. This way the predictor could better accommodate a changing workload.

Performance prediction for interactive episodes in the presence of periodic activity relies on rescaling to allow the predictor to adapt to the changing workload. Our previous studies have shown that even lightweight background activity, such as MP3 playback, extends the duration of perceptible interactive episodes by an average of 14%. This implies that performance factors predicted based on data without the background activity would underestimate the necessary performance. To alleviate this problem, when periodic activity is detected, the *totalTime* variable is set to 100ms and *totalPF* is recomputed based on the new value. While providing a reasonable initial prediction, this change allows new performance factor data to take hold quickly.

6.3 Simulation methodology

Our simulator is driven by traces collected using a modified Linux kernel (2.3.99-pre3) running on a Dell Precision workstation 410, with only one of the two Pentium II 450Mhz processors enabled (512M RAM). The software environment was Mandrake Linux 7 with Helix Gnome 1.2. The traces used in this study are the same as the uniprocessor traces used in Chapter 3. All benchmarks were run by a live user. While we have collected multiple runs in each configuration, in this paper we only use a single run for each simulation. We aimed to repeat each run with MP3 in the background as accurately as possible, but there are slight variations between the runs. All the significant events (e.g., mouse clicks, text entry) were performed in the same order during each benchmark run. However, the exact path of mouse movement (and therefore the interactive episodes corresponding to them) and the amount of time between events varies from one run to the other.

The traces include all significant OS events during the benchmarks execution: thread swap events, system calls, and task information (e.g. name, pid, etc.). These are the same ones as the uniprocessor traces in Chapter 3. Based on this information, our simulator can reconstruct the communication events between the tasks (which imply the synchronization points between them) and simulate the effects of performance scaling. The upside of our methodology is that we have the flexibility to investigate a wide set of architectural parameters. The downside is that the absence of actual hardware prevents us from measuring total energy consumption and from calibrating our results.

6.4 Energy and performance implications

Our aim was to develop a performance scaling technique that can guarantee that user perceived performance does not degrade below a user-settable level. A detailed microarchitecture-level power analysis is beyond the scope of this paper, however we can derive some estimates regarding the expected energy savings using a few simple assumptions.

The metric we use is the energy factor, which is the ratio of the energy used by the scaled workload divided by its predicted energy use at full performance. Equation 9 gives the energy factor formula for a given workload, assuming that the workload is divided into *n* pieces that execute at the scaled voltage (v_i) and frequency (f_i) for the scaled amount of time (t_i). *T* refers to the total execution time at full speed, while the *max* subscript refers to the maximum value of the given variable.

$$EnergyFactor = \frac{\sum_{i=1}^{n} v_i^2 f_i t_i}{v_{max}^2 f_{max} T} \qquad \text{(EQ 9)}$$

Our model focuses on the CPU alone and does not take the power consumption of other devices (such as memory and peripherals) into account.

6.4.1 Processor and power scaling model

Our performance model is based on assumptions from the Intel XScale microarchitecture [23]. Our traces were collected on a 450Mhz Pentium II based workstation, and we make the simplifying assumption that these traces correspond to the full-speed performance on each of the simulated models.

We assume that for each performance transition, there is a 20 microsecond pause, during which the processor does not execute any instructions. This pause is due to the time it take to resynchronize the PLLs for the changed frequency value. After this, the performance transition time—during which the voltage level is changed—is assumed to be 1 millisecond regardless of starting and ending performance level. During this time, we assume that the processor is executing instructions at the rate corresponding to the lower of the two performance levels, but energy is being consumed at the higher.

Model	Frequency (Mhz)						
	150	333	400	600	773	800	1000
XSBase	▨	1	1.1	1.3	1.5	▨	▨
XSA	0.75	1	1.1	1.3	1.5	▨	▨
XSB	0.75			1.2		1.4	1.75

The table shows the given frequency-voltage pairs of our models. Dark areas represent frequency levels that are not supported in a given model.

Table 6.1: Frequency-voltage pairs in our energy models

Table 6.1 shows the frequency-voltage values that we used in our power models. Equations 9, 10, and 11 show the voltage equations corresponding to each model. These were computed based on published data. The XSBase model corresponds closely to the high-end XScale part (80200M733) described in [23].

$$v_{XSBase} = -5\times10^{-8}f^2 + 0.0012f + 0.6261 \qquad \text{(EQ 10)}$$

However, since this model only has a 2.32x frequency range, we extended it to 5.15x by allowing it to go as low as 150Mhz in the XSA model.

$$v_{XSA} = -4\times10^{-7}f^2 + 0.0015f + 0.5324 \qquad \text{(EQ 11)}$$

The XSB models the parameters of a high-end device that is a research prototype. This processor can vary its performance between 150Mhz and 1000Mhz for a 6.67x swing.

$$v_{XSB} = 5\times10^{-7}f^2 + 0.0005f + 0.6624 \qquad \text{(EQ 12)}$$

In all our energy calculations, we assume that the OS power manager puts the processor into a low power sleep mode immediately when no instructions are executing. We do not attribute a power cost to this operation and assume that it happens instantaneously.

During our evaluation we specify a quantization factor for each of the power models. On an actual processor not all frequency/voltage pairs can be directly set up, one must choose from a set of predetermined values. This means that when the performance estimator requests a given performance setting, the actual performance value is rounded up to the next quantum. In our experiments we mostly focus on models where frequency is quantized at 5% steps (100%, 95%, 90%, etc.). We denote quantized models with the suffix ‘q’ followed by the quantum size.

6.5 Performance and energy characteristics

In this section we examine the characteristics of the basic performance setting algorithm and propose some improvements. Our evaluation focuses on the following three main goals:

- Minimizing the number of performance-level transitions.
- Minimizing the amount of increase in the duration of perceptible interactive episodes.
- Maximizing energy reduction.

These three aspects are closely interrelated. Reducing the number of performance-level transitions is important, because each transition has a delay and energy cost that negatively affects both the perceptible performance and energy savings. On the other hand, increasing the interactive episode duration has a positive effect on energy savings, because the longer interactive episodes may stretch, the slower they can run. However, this has a negative impact on the user perceived performance. While the increase in perceptible interactive episode duration in all cases falls within the acceptable range (since ensuring this is part of the performance-setting algorithm’s job, see Section 6.2.4), we seek to minimize it, i.e. our methodology favors faster response time over energy savings.

The perceptible interactive episode-length increase is computed for all scaled episodes that fall above the perception threshold by dividing the scaled episode length by either the full-speed episode length or the perception threshold, depending on whether the original episode length was longer or shorter (respectively) than the perception threshold.

Table 6.2 shows our baseline results using the XSB model (without quantization) and 50ms perception threshold and assumptions described in Section 6.4.1. The mean perceptible interactive episode length increase in all cases is under 30%. Applications that have many short episodes (e.g. Xemacs and Netscape) tend to have the largest increase, while workloads with long episodes (e.g. Ghostview and GIMP) exhibit the smallest increase. This makes sense given that our acceptable delay function (*PanicThreshold*) allows more performance degradation for shorter episodes than for longer ones. One

Benchmarks	No MP3 in background				MP3 playback in background			
	Perf. trans.	Mean perceptible IE length increase	Median perceptible IE length increase	Energy factor	Perf. trans.	Mean perceptible IE length increase	Median perceptible IE length increase	Energy factor
Acrobat Reader	543	13%	7%	0.91	668	13%	11%	0.84
FrameMaker	155	20%	11%	0.89	191	9%	7%	0.75
Ghostview	510	5%	1%	0.98	1149	5%	1%	0.91
GIMP	919	5%	4%	0.97	1731	5%	4%	0.91
Netscape	1026	18%	14%	0.87	1739	21%	12%	0.82
Xemacs	381	23%	20%	0.30	1417	29%	33%	0.34

Table 6.2: Performance characteristics (XSB, 50ms perception threshold)

should also note that the number of performance transitions increases significantly (up to four times) when MP3 playback is running in the background. This is because, when a periodic episode is running, the performance setting algorithm alternates between the setting for interactive episodes and the setting for the periodic task. The energy factor tends to be lower when MP3 is running than without. The reason for this is that in most cases the MP3 player requires a lower performance setting than the interactive application. Xemacs is an exception: the benchmark's interactive episodes require a lower performance-setting than the MP3 player.

Table 6.3 illustrates the effects of quantization on the results. This model corresponds more closely to an actual hardware implementation than the one used in the previous table. Quantization tends to slightly reduce energy savings and also reduce perceptible interactive episode lengths. The only benchmark, where this was not the case is Xemacs, where quantization corrects a few mispredictions, causing both an increase in energy savings and a decrease in the average perceptible episode length.

Perhaps the most striking improvement over the data in Table 6.2 is the dramatic reduction in the number of performance-level transitions (in some cases more than 300-fold) when no MP3 playback is running concurrently with the interactive application. This behaviour is also due to the fact that when there is no periodic background activity, the successively predicted performance levels are close to each other, causing quantization to eliminate the minor corrections. When MP3 playback is present, the deliberate transitions between interactive and periodic modes keeps the number of transitions high.

Benchmarks	No MP3 in background				MP3 playback in background			
	Perf. trans.	Mean perceptible IE length increase	Median perceptible IE length increase	Energy factor	Perf. trans.	Mean perceptible IE length increase	Median perceptible IE length increase	Energy factor
Acrobat Reader	28	12%	5%	0.92	664	12%	10%	0.86
FrameMaker	15	17%	11%	0.90	184	7%	6%	0.77
Ghostview	10	4%	0%	0.99	1135	4%	0%	0.92
GIMP	28	4%	3%	0.98	1533	5%	3%	0.92
Netscape	32	17%	12%	0.88	1547	20%	11%	0.84
Xemacs	15	15%	14%	0.26	1416	29%	31%	0.32

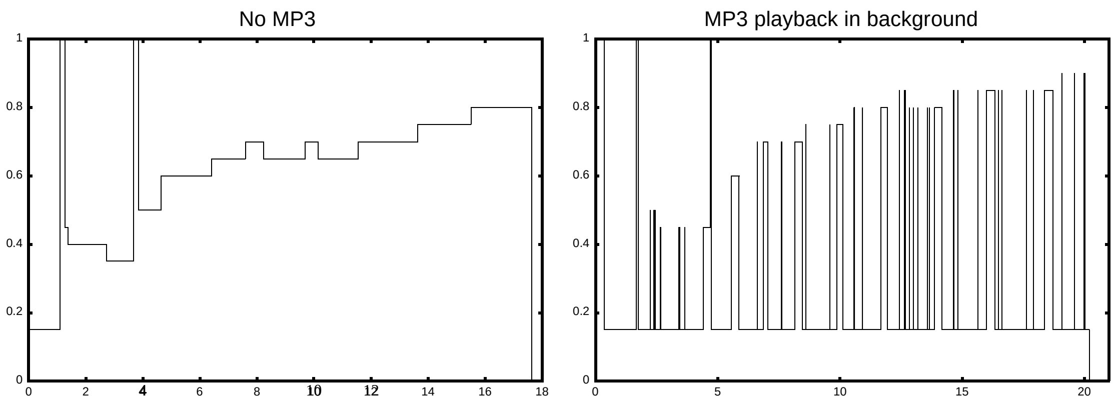
Table 6.3: Performance characteristics (XSBq5, 50ms perception threshold)

Benchmarks	MP3 IEPerf Transition start latency = 1ms				MP3 IEPerf Transition start latency = 5ms			
	Perf. trans.	Mean perceptible IE length increase	Median perceptible IE length increase	Energy factor	Perf. trans.	Mean perceptible IE length increase	Median perceptible IE length increase	Energy factor
Acrobat Reader	637	11%	4%	0.86	125	13%	6%	0.75
FrameMaker	153	8%	7%	0.76	81	12%	9%	0.73
Ghostview	1031	5%	1%	0.91	222	6%	1%	0.86
GIMP	854	5%	4%	0.88	334	6%	6%	0.83
Netscape	1072	18%	12%	0.83	340	20%	14%	0.72
Xemacs	1047	29%	32%	0.32	980	34%	39%	0.31

Table 6.4: Performance characteristics with start latency (XSBq5, 50ms perception threshold)

The number of performance transitions can be further reduced based on an observation of Figure 6.2. We have pointed out that the majority of interactive episodes are very short (less than 1ms) and that very little time is spent in those episodes (<5%). When there is no periodic background activity, the effect of the short interactive episodes is negligible since the performance level is always set at the predicted interactive performance level, however when there is an MP3 player in the background, these short interactive episodes cause an unnecessary transition from the periodic to the interactive performance level. When the interactive episode is very short, this transition simply wastes energy, since the episode is likely to be finished before the transition is over.

This observation suggests a strategy that waits for a certain amount of time before starting a transition to the interactive performance level. Table 6.4 illustrates the effects of



Two runs of the Acrobat Reader benchmarks are shown side by side with and without MP3 playback during the run. Perception threshold was set to 200ms, and data was generated using the Enhanced predictor and XSBq5 model. Spikes to full performance represent instances when the PanicThreshold was reached. Note that the graphs do not show transitions to sleep mode.

Figure 6.6: Performance-setting using the Enhanced predictor (XSBq5 model)

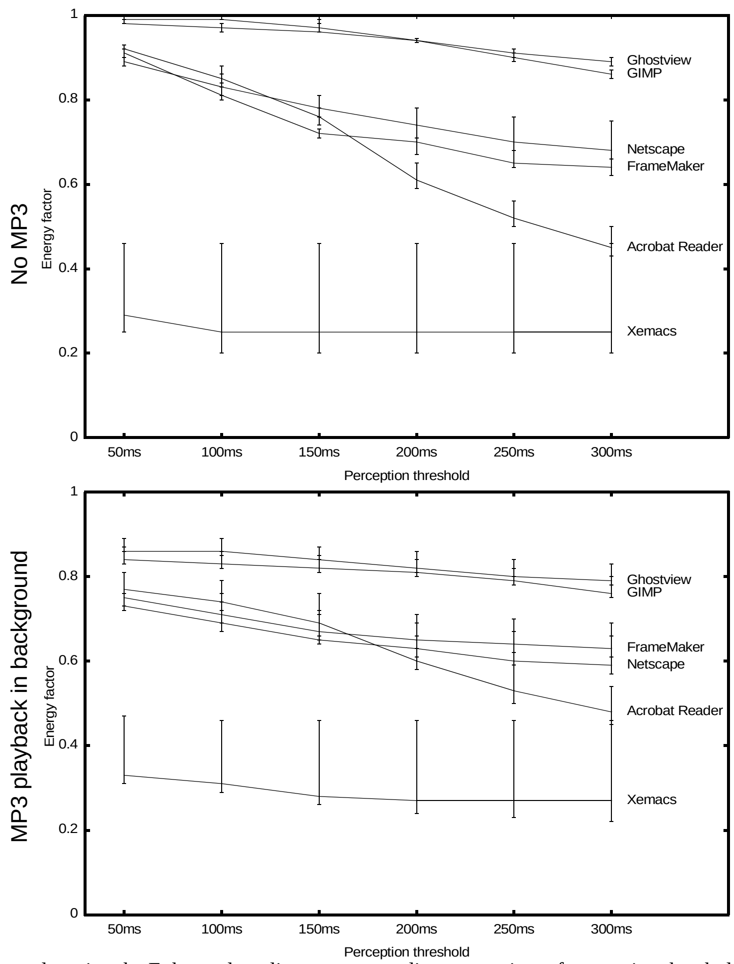
a transition-start latency before interactive episodes. The data is only shown for the benchmarks with MP3 playback in the background, because there was no significant change in results when background activity was not present.

Contrasting with Table 6.3 shows that a 1ms transition-start latency leaves the energy factors mostly unchanged but causes a small reduction in the number of performance transitions. Extending the transition-start latency to 5ms causes both a significant reduction in energy consumption and in the number of performance transitions (up to a 5-fold reduction). Moreover, the average perceptible interactive episode-lengths stay at around the same level as before.

6.5.1 The Enhanced predictor

The previous section suggests two minor changes to the Basic predictor: 1) quantizing the allowable performance levels and 2) waiting for a certain time before changing the performance factor when an interactive episode starts. For simplicity’s sake we only use a statically specified transition start latency of 5ms in the Enhanced predictor. A more sophisticated predictor could dynamically compute a per-process value based on the distribution of episode lengths.

Figure 6.6 shows the performance factors of the Enhanced predictor during the execution of the Acrobat Reader benchmark. Contrasting the figure with Figure 6.5



These graphs show results using the Enhanced predictor corresponding to a variety of perception thresholds. At each perception threshold level, we show the energy factors for the QSBaseQ5 (top point), XSAq5 (middle point, connected) and XSBq5 (bottom point) models.

Figure 6.7: Energy factors for different perception thresholds and models.

reveals the reduced number of performance changes and fewer transitions to full performance due to reaching the PanicThreshold.

Figure 6.7 shows the energy factors using the Enhanced predictor and the XSBaseQ5, XSAq5 and XSBq5 power models given a variety of perception thresholds. Our results show that while on the measurement machine there is little opportunity for power savings, as the peak performance of the processor gets faster, energy savings will be more pronounced. We must note that our traces were collected on a 450Mhz Pentium II machine, and today's high-end processors are already 2-3 times faster. We estimate that the energy factor on today's high-end desktops could be in the 10%-75% range at the 50ms-100ms perception threshold.

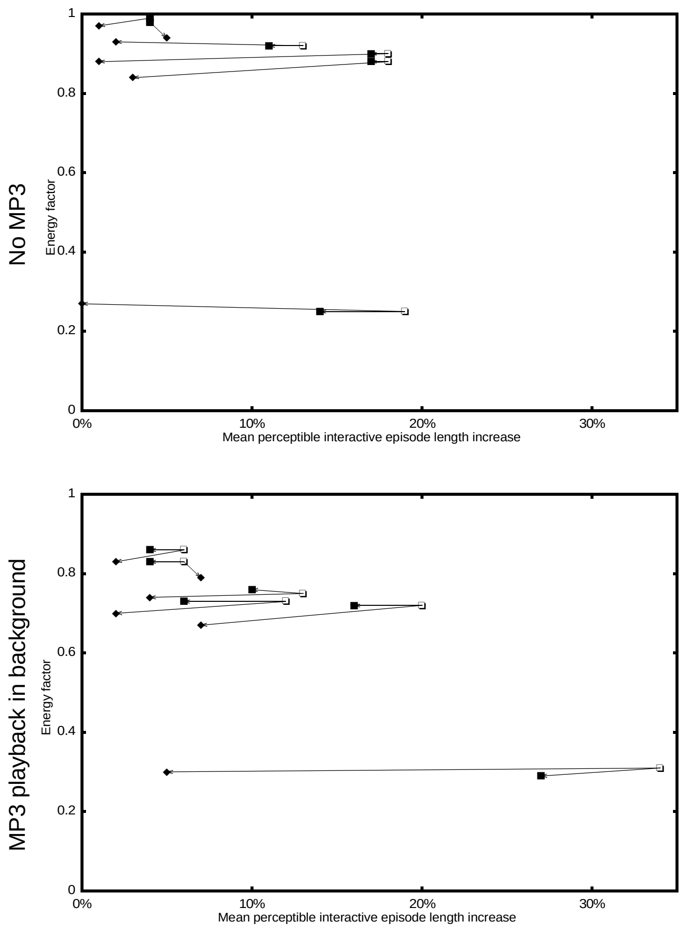
In the figure, energy factors corresponding to the XSAq5 model are connected by a line at each perception threshold, the results for the XSBaseQ5 and XSBq5 are shown as the bars above and below each point (respectively). In all cases, XSBq5 achieves the largest energy savings, while XSBaseQ5 achieves the least. For most applications the difference between the three models is small: in most cases less than 5%. However, the difference is significant on the Xemacs benchmark runs, since this application spends most of its time at the lowest performance setting allowed in each model. In this case, the lower minimum performance levels of the XSA and XSB models give them a significant edge.

Periodic activity has the effect of vertically compressing the graph towards its center. The load due to the background activity increases the energy savings for benchmarks with long interactive episodes. On the other hand, one can observe the exact opposite effect when the interactive episodes are short (Xemacs).

Figure 6.7 contrasts the energy factors and effects on interactive episode lengths of three different models. The three connected datapoints for each benchmark correspond to the Enhanced predictor, an aggressive hardware model combined with the Enhanced predictor (see Section 6.5.2) and the Oracle predictor (where the predictor has perfect information, but the hardware model remains realistic). Points that are closer to the lower left corner of the graph are better. Since the maximum increase of interactive episode lengths is bound by the PanicThreshold (see Equation 8), the increase shown in the graphs is not detrimental to the user experience. However, our goal is to minimize the interactive episode length increase for a given energy factor.

The Oracle predictor differs from the Enhanced predictor in that it has exact knowledge of the lengths of interactive episodes as soon as they begin. For this reason, there is no need to introduce a 5ms latency before transitioning to a new performance level (see Section 6.5 and Table 6.4 for why this latency is necessary in the Enhanced predictor). The detection of periodic episodes works the same way as in the enhanced predictor. The model used for the data is XSBq5 with 50ms perception threshold.

The figure shows that while there is a small reduction of the energy factors, the primary difference between the Enhanced and the Oracle predictors is that the latter has a significantly smaller average increase of interactive episode lengths. In fact, the increase



The graphs show how the energy factors and response times change when an Oracle predictor is used (◆) or the latencies due to performance and voltage changing ar eliminated (■) compared to the realistic Enhanced predictor with the XSBq5 model (❑) with 50ms perception threshold. The datapoints for each benchmark are connected. Benchmarks from top to bottom: Ghostview, GIMP, Acrobat Reader, FrameMaker, Netscape, Xemacs.

Figure 6.8: Energy factor and response time improvements

is due only to the latencies inherent in the power model, not to prediction error. If the Oracle predictor was allowed to extend the interactive episodes by the same amount as the Enhanced predictor, one would see an additional 10%-20% percent increase in energy savings on benchmarks that have not already reached the minimum performance-level of the processor. Perhaps one way of getting closer to the prediction accuracy of the Oracle predictor would be to base performance scaling decisions on the expected and observed statistical distribution of episode lengths. However, for such a scheme to work well, we believe that it is important to be able distinguish episode instances from one another, not just episode types.

6.5.2 Desired hardware improvements

We have already shown that quantization can have a positive effect on both energy savings and interactive performance. In our measurements we found that the quantum size does not greatly impact our results. Using the Enhanced predictor, the difference in energy savings due to quantum sizes of 5%, 10%, and 20% are negligible. While, in all cases the 5% quantum size has the smallest energy factor, the 20% quantum size is only behind by at most 1%. On the Oracle predictor, the difference in energy savings due to different quantum sizes, is always under 5%.

For all our measurements it is assumed that there is a 20 microsecond pause when a performance transition is initiated and that a transition takes 1 millisecond. The current pause duration, which we believe is indicative of what can realistically be expected, is short enough and thus eliminating it has neither a significant impact on energy savings, nor on perceptible interactive performance. However, there might be other reasons for lowering it, such as latencies incurred during communication with peripheral devices.

Reducing the lengths of performance transitions, on the other hand, has a positive effect on both the perceptible performance and energy savings. Figure 6.7 contrasts a model without any latencies for performance transitions with both a model based on realistic assumptions and one with an Oracle predictor. While shortening performance transitions has an overall positive effect, a better prediction mechanism (as shown by the Oracle predictor) could achieve even more significant improvements. We believe that the Enhanced predictor could be improved by giving it the ability to distinguish between episode instances, not just episode types.

6.6 Conclusions

We show how an automatic episode detection mechanism can be used to guide the performance-setting decisions for a processor that supports dynamic performance and voltage scaling. This system can derive and predict episode deadlines automatically, without the need to modify existing user programs. We have shown that our approach can achieve significant energy savings while ensuring that interactive performance stays at an acceptable level. We are currently working on the evaluation of our algorithms on a system that is capable of dynamic voltage scaling.

While our methodology does not require modification of user programs, an optional API might be useful for applications that want to take full advantage of performance-setting. One of the biggest shortcomings of the current predictor is its inability to distinguish episode instances from one another. An API that allows the programmer to delineate and name critical episodes while optionally specifying its type and deadline would help. The API might consist of the following system calls:

```
episode_begin <id> [type] [deadline]
episode_end <id>
```

The id is a per-task identifier assigned by the programmer to distinguish one episode from another. The type field optionally categorizes the episode into interactive, periodic, and producer or consumer categories. The deadline field optionally specifies the maximum length of the episode. The idea behind this API is that its main role is to give hints to the existing prediction and communication-tracking mechanism, instead of superseding it. For example, there is no need to specify dependencies between episodes since that information can be derived automatically from information in the kernel. The main use of this API would probably be in commonly used libraries (e.g. Xlib or gtk), which would allow most off-the-shelf programs to use the API without program modifications. The use of the proposed API in libraries would be a foundation on which more sophisticated prediction algorithms could by layered. Being able to distinguish between episodes instances, not just episode types would allow the use of more sophisticated history based algorithms for predicting the expected episode lengths.

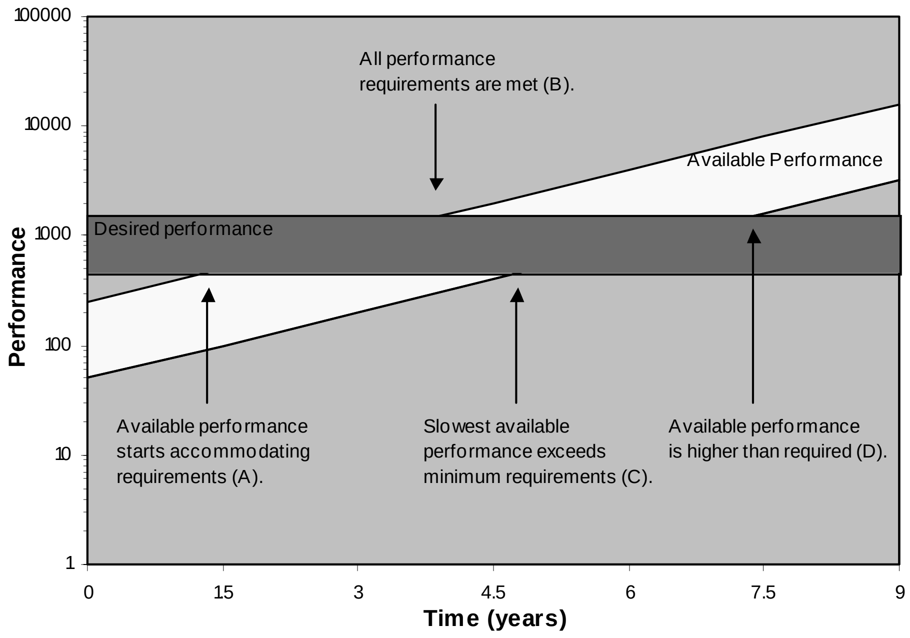
6.6.1 The performance gap

As the peak performance of processors improve, it is also important to allow dynamically voltage-scaled processors to accommodate an increasingly wider range of performance levels. While many of our benchmarks could take advantage of higher peak performance, the Xemacs benchmark could run well at even lower settings than the minimum performance-level afforded by our processor models. Further performance improvements are necessary to improve the user experience of high-end applications. However, the performance of an increasing number of workloads are fast enough, yet these applications are getting faster as well due to the need to improve high-end applications.

Running adequately performing applications faster than necessary is wasteful since higher performance comes at a significant energy cost. Current production processor designs that allow dynamic voltage scaling only allow a maximum of five-fold change in frequency. The experimental lpARM processor allows for a 10 fold variance in performance [40], which may indicate that with sufficient emphasis on voltage scaling, a wider range of performance levels can be attained. As processors are developed to be higher performance, the performance in their slowest setting also increases, meaning that more and more applications will run faster than required even in the slowest processor setting. Assuming that current trends of quadrupling performance every three years hold true and that the performance requirements of existing applications do not increase over time, applications have a three year window in which their performance can be scaled. Outside this window they either run at the minimum or maximum performance levels of the processor.

A workload's performance requirements may change with time, since new revisions may incorporate new features or the operating system can incur new functionality under the hood that impacts the performance of the applications running on it. For example, the OS four years from now may include a more sophisticated rendering engine or a refactored API that is more convenient for programmers, but one that also increases the performance requirements of existing programs. However, the reverse can also be true: a future OS may actually have lower overhead due to optimizations. In the approximately two years that elapsed since the beginning of the work presented in this dissertation, successive versions of our benchmarks did not show an increased appetite for processor performance. In fact, our experience with thread-level parallelism indicates that the Linux kernel has been becoming more efficient over the same time period (see Section 5.2).

Figure 6.9 illustrates how a hypothetical application's desired performance is met by a processor over time. Before point A, the workload cannot be run adequately on the processor. After point A, the needs of the workload are beginning to be met by the hardware and at point B, even the peak performance requirements of the application are accommodated. The application takes best advantage of the hardware between points B



The figure illustrates how the available performance increases quadratically with time. Desired performance of individual applications, on the other hand, stays relatively constant with time. The intersection points of the two areas represent significant milestones during the evolution of a processor and workload.

Figure 6.9: Desired and available performance

and C. Here, the hardware meets all processing requirements and can be slowed down enough to exactly meet the minimum requirements. After point C, even the minimum performance setting of the processor is getting to be more than what's needed. Beyond point D there is no need to run the processor above its minimum speed at all. We have chosen the requested performance range only for illustration purposes, it does not represent an actual application. A real workload would most likely have a wider requested performance range, with the minimum being significantly lower than the one in the picture. For these workloads points B and D are the most significant.

There is only a point to reducing performance if it implies advantages in other areas, such as requiring less power. This aim can be accomplished by developing circuit techniques that allow a wider trade-off between high performance and low-energy operating modes, allowing the processor to run slower and saving energy while also enabling it to run fast when required. Alternatively, one could address the problem at the microarchitecture level by reducing the level of speculation and selectively turning off processing units or by using multiple processing cores that are tuned to different levels of performance. Such an approach could be used both for uniprocessor and multiprocessor systems.

6.6.2 The role of multiprocessing

Multiprocessing is primarily viewed as a means of reaching higher performance. Our results in Chapter 5 indicate that using two processors instead of one can significantly improve the interactive response time of desktop applications. We also argue that more than two processors are unlikely to greatly improve performance on current desktop workloads. Aside from increasing performance, can multiprocessing be also used to increase power savings?

One way of looking at the effects of multiprocessing is to view it as a single processor that just happens to run faster (by exploiting thread-level parallelism). The system is designed to be symmetrical, both processors run at the same speed. Software can adjust the speed setting of the system, which scales the individual processors equally. In this case, the multiprocessor effectively brings the future a little closer by allowing an application to run as fast today as it would run in a few months or years time. The multiprocessor could conceivable save power since by exploiting thread-level parallelism, it effectively would allow each processor to run slower than they could individually. Whether this is feasible depends on the following:

- *How much more the processor can be slowed down.* If it is already running in the slowest mode, then there is no need for multiprocessing. If the applications run close to the minimum speed (how close depends on the parallelism of the application), then all the benefits of further speed reductions may not be realized.
- *The expected parallelism of the applications.* Our results show that the average parallelism of our benchmarks are around 1.3 (30% of the time both processors in the system run tasks concurrently). The expected amount of concurrency needs to be factored in to figure out whether a multiprocessor design is competitive with a uniprocessor from a power perspective. In [43] an argument is made that an SMT design is more energy efficient than a uniprocessor under workloads that supply ample amounts of parallelism. Further study is required to show what level of parallelism is the break-even point.

A further refinement is to set the performance level for each CPU separately. This approach has a more complicated software component for figuring out the desired performance level, however the power savings can be greater. As a first step, the performance-setting algorithm still needs to figure out how long an episode will take and based on this information come up with an overall speed factor. Once the application starts running, further speed adjustments can be made on the individual CPUs to reduce power consumption. In our experience most concurrency is asymmetrical; one processor has almost always more work to do than the other. It is not a requirement that communications between tasks be unidirectional for asymmetrical performance scaling. The only requirement is that the communicating tasks should not be blocked waiting for each other while the other is executing.

An interesting design point for a low-power multiprocessor would be to consider processing cores that have been optimized for different levels of performance. One of the processors could provide high performance, while the other could be optimized for lower performance and lower energy requirements. It is conceivable that a system in which one core would run at about half of the maximum performance of the other, would still allow almost all the benefits of multiprocessing at the high-end (since concurrency of desktop applications is on average around 1.3, less than the 1.5 such a system could accommodate), while also allowing a greater range of performance settings overall. While, both processors would still need to have the ability to adjust their performance levels (voltage and frequency), however such a system would cover a broader range than a uniprocessor that is optimized for high performance at peak power.

Chapter 7
Conclusion

This dissertation shows that multiprocessing can be used effectively to improve the response-times of long interactive episodes and that our performance-setting algorithm applied to dynamic voltage scaling can yield significant power savings when response-times are fast enough. We make the following research contributions in this dissertation:

A metric and portable methodology for measuring concurrency

We developed a metric and a technique for measuring thread-level parallelism (TLP). We found that machine utilization does not accurately measure the concurrency in the system, because the large amount of idle time during interactive benchmarks obscures the presence of concurrent execution. TLP is defined as the machine utilization over the non-idle portions of the benchmark's execution. This definition sidesteps the problems of the machine utilization metric and allows us to use more realistic workloads. Intuitively, TLP reflects the speedup due to concurrent execution on the non-idle portions of the workload. We also developed a portable technique for measuring TLP, and used it to measure the concurrency of a broad set of applications across three operating systems: BeOS, Linux, and Windows NT. We found that TLP is very workload and operating system dependent and that while most applications consist of many threads, most often these threads do not run concurrently.

Monitoring interactive applications

The key performance metric of interactive applications is not overall throughput but response time. However, due to the difficulty of isolating execution episodes that have direct bearing on response time, even benchmarks for interactive applications (e.g. Sysmark [46] and Winstone [49]) only measure end-to-end throughput. The results of these benchmarks can be misleading because they ignore two key properties of interactive applications: not all parts of the program's execution are equally important and faster response-time is not always better. There is a point beyond which improvements are imperceptible to the user, and, thus are unnecessary. To gain insight into the behavior of interactive applications, we developed a technique that provides automatic feedback to the operating system about the interactive performance of executing programs. This methodology automatically quantifies response time by monitoring the communication between executing threads (both system and application) and by observing the applications' interactions with the kernel through system calls. This mechanism is implemented in the Linux kernel and requires no modification of user programs.

Impact of multiprocessing on interactive performance

Multiprocessing is already prevalent in servers where multiple clients present an obvious source of thread-level parallelism. However, the case for multiprocessing is less clear for desktop applications. Nevertheless, architects are designing processors that count on the availability of thread-level parallelism. As a reality check, we applied our monitoring technique to measure the impact of multiprocessing on interactive performance. The results show that running our benchmarks on a dual-processor machine improves response times of perceptible events by as much as 36% and 22% on average—out of a maximum possible 50%. The benefits of multiprocessing are even more apparent when background tasks are considered. Running a simple MP3 playback program in the background degrades response times by an average of 14% on a uniprocessor vs. only 4% on a dual-processor machine. While four processors would be underutilized, dual-processor machines provide an effective way of improving response times.

Performance-setting for dynamic voltage scaling

Our investigations into the effects of multiprocessing show that, while there is still a need to improve processor performance to reduce response-times of a few long-running episodes, the majority of interactive episodes—even on a uniprocessor—fall under the human perception threshold. When response-times are fast enough for further improvements to be imperceptible, the processor can be slowed down to save energy. Some contemporary processors (e.g. Transmeta Crusoe [25] and Intel XScale [23]) support dynamic voltage and frequency scaling which allows one to make fine granularity trade-offs between power-use and performance, provided there is a mechanism for controlling that trade-off. We developed an algorithm, based on the combination of the episode detection technique and a performance-level predictor, for automatically controlling dynamic voltage scaling to optimize energy use. Unlike previous automated approaches, this method works equally well with irregular and multiprogrammed workloads. Moreover, it has the ability to ensure that the quality of interactive performance is within user-specified parameters. Experiments show that as a result of this algorithm, processor energy savings of as much as 75% can be achieved with only a minimal impact on the user experience.

7.1 Directions for future research

We have shown that even a simple performance-setting algorithm can be effective at reducing the performance level of dynamically voltage scaled processors. However, a comparison with the Oracle predictor shows that there is still room for improvement. We believe that the biggest improvement for prediction accuracy would come from enabling the predictor to distinguish between episode instances, not just episode types. For example, it might be beneficial to know when an interactive episode was started as a result of the same button press as before to increase the prediction correlation between similar interactive events. Distinguishing episode instances is not difficult to do by modifying the GUI libraries or applications, but it is challenging to do automatically. Section 5.5 outlines an API that, given suitable modifications of programs and libraries, could help to solve this problem. The use of the proposed API in libraries would be a foundation on which more sophisticated prediction algorithms could by layered.

While our results for the improvement of interactive performance due to multiprocessing were all measured on an actual machine, the energy savings, resulting from our performance-setting algorithm, were only simulated. When systems based on processors supporting dynamic voltage scaling become available, we intend to perform a real-world evaluation of our algorithms. These measurements will be able to quantify how the power consumption of the entire system is affected, and allow us to better evaluate the impact of our algorithms on the user-perceived response times.

The core idea of the episode detection technique—on-line monitoring and dynamic adaptation—could be applied to other algorithms. For example the scheduler could be made aware of which tasks interact with the user and could use this information to further improve interactive performance. Perhaps this idea could be generalized to derive deadlines for repetitive real-time tasks automatically.

This dissertation has shown that multiprocessing can be used effectively to improve the response-times of long interactive episodes and that dynamic voltage scaling can yield significant power savings when response-times are fast enough. Can these two ideas be combined to yield a power-efficient multiprocessor?

Bibliography

[1] ACPI 2.0. http://www.teleport.com/~acpi/

[2] Apple press release: Apple Debuts New PowerMac G4s with Dual Processors. http://www.apple.com/pr/library/2000/jul/19g4.html

[3] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Berghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. *Proceedings of the 27th Annual International Symposium on Computer Architecture,* June 2000.

[4] L. Benini, A. Bogliolo, S. Cavallucci, and B. Ricco. Monitoring System Activity for OS-directed Dynamic Power Management. *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '98),* August 1998.

[5] S. K. Card, T. P. Moran, and A. Newell. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum Associates, Publishers, 1983.

[6] J. B. Chen, Y. Endo, K. Chan, D. Mazieres, A. Dias, M. Seltzer, and M. D. Smith. The Measured Performance of Personal Computer Operating Systems. *Proceedings of the 15th ACM Symposium on Operating System Principles,* pp. 299-313, December 1995.

[7] E. Cota-Robles, J. P. Held, "A Comparison of Windows Driver Model Latency Performance on Windows NT and Windows 98" *3nd Symposium on Operating Systems Design and Implementation,* February, 1999.

[8] M. Culbert. Low Power Hardware for a High Performance PDA. *Proceedings of the Thirty-Ninth IEEE Computer Society International Conference,* March 1994.

[9] K. Diefendorff. Power4 Focuses on Memory Bandwidth: IBM Confronts IA-64, Says ISA Not Important. Microprocessor Report, Volume 13, Number 13, October 6, 1999.

[10] K. Diefendorff. Compaq Chooses SMT for Alpha: Simultaneous Multithreading Exploits Instruction- and Thread-Level Parallelism. Microprocessor Report, Volume 13, Number 16, December 6, 1999.

[11] Y. Endo, Z. Wang, J. B. Chen, and M. I. Seltzer. Using Latency to Evaluate Interactive System Performance. *2nd Symposium on Operating Systems Design and Implementation,* pp. 185-199, October 1996.

[12] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. Thread-level parallelism of desktop applications *Proceedings of Workshop on Multi-threaded Execution, Architecture and Compilation*, Toulouse, France, January 2000.

[13] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. Thread-level parallelism and interactive performance of desktop applications. *Proceedings of Conference on Architectural Support for Programming Languages and Operating Systems*

(ASPLOS-IX), November 2000.

[14] K. Flautner, S. Reinhardt, and T. Mudge. Automatic Performance-Setting for Dynamic Voltage Scaling. *Submitted to the International Conference on Mobile Computing and Networking (MOBICOM-7),* July 2001.

[15] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. *Proceedings of 17th ACM Symposium on Operating Systems Principles (SOSP-17),* December 1999.

[16] R. Golding, P. Bosch, and J. Wilkes. Idleness is Not Sloth. HP Labs Technical Report HPL-96-140, October 1996.

[17] K. Govil, E. Chan, and H. Wasserman. Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU. *Proceedings of the First International Conference on Mobile Computing and Networking*, November 1995.

[18] D. Grunwald, P. Levis, K. Farkas, C. B. Morrey III, and M. Neufeld. Policies for Dynamic Clock Scheduling. *Proceedings of the Fourth Symposium on Operating Systems Design & Implementation*, October 2000.

[19] L. Hammond and K. Olukotun. *Considerations in the Design of Hydra: a Multiprocessor-on-a-Chip Microarchitecture.* Stanford University Technical Report No. CSL-TR-98-749.

[20] C. Hauser, C. Jacobi, M. Theimer, B. Welch, and M. Weiser. Using Threads in Interactive Systems: A Case Study. *Proceedings of the 14th ACM Symposium on Operating Systems Principles,* pp. 94-105, December 1993.

[21] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-Directed Dynamic Frequency and Voltage Scheduling. *Proceedings of the Workshop on Power-Aware Computer Systems (PACS ’00),* November 2000.

[22] Intel. Intel StrongARM SA-1110 Microprocessor Developer’s Manual. http://developer.intel.com/design/strong/manuals/278240.htm

[23] Intel. Intel 80200 Processor Based on Intel XScale Microarchitecture. http://developer.intel.com/design/iio/manuals/273411.htm

[24] C. M. Krishna and Y-H Lee. Voltage-Clock-Scaling Adaptive Scheduling Techniques for Low Power Hard Real-Time Systems. *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000), 2000.*

[25] D. Laird, Crusoe Processor Products and Technology http://www.transmeta.com/press/download/pdf/laird.pdf, January 2000.

[26] D. Lee. Energy Management Issues for Computer Systems. http://www.cs.washington.edu/homes/dlee/frontpage/mypapers/generals.ps.gz

[27] D. C. Lee, P. J. Crowley, J. Baer, T. E. Anderson, and B. N. Bershad. Characteristics of Desktop Applications on Windows NT. *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[28] B. Lewis, D. J. Berg, *Multithreaded programming with pthreads*. Sun Microsystems, 1998.

[29] J. Lo, L. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[30] J. R. Lorch and A. J. Smith. Reducing Processor Power Consumption by Improving Processor Time Management in a Single-User Operating System. *Proceedings of the Second Annual International Conference on Mobile Computing and Networking,* November 1996.

[31] Y.-H. Lu, T. Simunic, and G. De Micheli. Software Controlled Power Management. *Proceedings of the 7th International Workshop on Hardware/Software Codesign,* October 1997.

[32] *Microprocessor Architecture for Java Computing*. http://www.sun.com/microelectronics/MAJC, Sun Microsystems, 1999.

[33] J. Monteiro, S. Devadas, P. Ashar, and A. Mauskar. Scheduling Techniques to Enable Power Management. *Proceedings of the 33rd Annual Conference on Design Automation (DAC-33),* 1996.

[34] T. Mudge. Power: A First Class Design Constraint for Future Architectures. *Proceedings of the 7th International Conference on High Performance Computing (HiPC 2000),* December 2000.

[35] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-aware Adaptation for Portable Computer Energy Management. *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16),* October 1997.

[36] T. Okuma, T. Ishihara, and H. Yasuura. Real-Time Task Scheduling for a Variable Voltage Processor. *Proceedings of the International Symposium on System Synthesis*, November 1999.

[37] D. R. Olsen. *Developing User Interfaces.* Morgan Kaufmann Publishers, 1998.

[38] T. Pering, T. Burd, and R. Brodersen. The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms. *Proceedings of International Symposium on Low Power Electronics and Design 1998*, pp. 76-81, June 1998.

[39] T. Pering, T. Burd, and R. Brodersen. Dynamic Voltage Scaling and the Design of a Low-Power Microprocessor System. *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1998.

[40] T. Pering, T. Burd, and R. Brodersen. Voltage Scheduling in the lpARM Microprocessor System. *Proceedings of the International Symposium on Low Power Electronics and Design 2000,* July 2000.

[41] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic Voltage Scaling on a Low-Power Microprocessor. UbiCom-TechhicalReport/2000/3, Delf University of Technology, The Netherlands, 2000.

[42] D. A. Solomon, *Inside Windows NT Second Edition.* Microsoft Press, 1998.

[43] J. S. Seng, D. M. Tullsen, and G. Z. N. Cai. Power-Sensitive Multithreaded Architecture. *Proceedings of International Conference on Computer Design 2000*, September 2000.

[44] Y. Shin and K. Choit. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. *Proceedings of the 36th Annual Design Automation Conference, 1999.*

[45] SPEC 2000. http://www.spec.org

[46] Sysmark 98. http://www.bapco.com/sys98k.htm

[47] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-chip Parallelism. *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 206-218, June 1995.

[48] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. *Proceedings of the First Symposium of Operating Systems Design and Implementation*, November 1994.

[49] Winstone 99. http://www.zdnet.com/zdbop