

ABSTRACT

SMITH, DANIEL M. Asymmetric Task Scheduling on Simultaneous Multithreading Processors. (Under the direction of Professor Vincent W. Freeh).

The performance of a simultaneous multithreaded CPU is greatly impacted by the behavioral characteristics of the threads competing for resources during concurrent execution. Most of the research aimed at improving SMT performance, or characterizing beneficial workload mixes, has targeted a multi-process parallel computation environment. Even in cases where the thread mix was heterogeneous, the CPU contexts were still viewed as two semi-independent resources, both of which were unbiased in their task selection.

We investigate an alternative method for operating system designers to utilize an SMT CPU. By confining user processes to a single context of the CPU and allowing kernel tasks to utilize the other context when necessary, we are able to, in many cases, provide better application performance than either an equivalent uniprocessor system, or an SMT system that is being treated as an SMP. In addition to operating in this special mode, an operating system may also choose to alternate between it and a conventional multiprocessing configuration, depending on which provides better performance.

A modification to the Linux 2.6 kernel to achieve this desired behavior is presented, as well as quantitative results of SPEC benchmarks which show where our modification improves performance. We also demonstrate how our modifications are sufficiently transparent to allow conditional mode selection at runtime.

Asymmetric Task Scheduling on Simultaneous Multithreading Processors

by

Daniel M. Smith

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science

Department of Computer Science

Raleigh

2005

Approved By:

Dr. Vincent W. Freeh
Chair of Advisory Committee

Dr. Frank Mueller

Dr. Jun Xu

For my wife, Taylor

Biography

Daniel M. Smith was born on August 17th, 1981 in Chapel Hill, North Carolina, USA. He received a B.S. in Computer Science from Appalachian State University in Boone, North Carolina in May 2003.

In August 2003, he entered the Masters program in Computer Science and North Carolina State University in Raleigh, NC. He joined the Operating Systems Research group under the direction of Dr. Vincent W. Freeh in August 2004, and was awarded the M.S. degree in May 2005.

Acknowledgements

I would like to thank my parents, Philip and Constance Smith, for supporting me throughout my academic career. Without their support (both moral and financial), my success would not have been possible.

I would also like to thank my thesis committee for their help and support in the process of preparing this work.

Contents

List of Figures	vii
List of Tables	viii
1 Introduction	1
1.1 Motivation	1
1.2 Contributions of this Thesis	2
1.3 Organization of this Thesis	3
2 Background	4
2.1 Symbiotic Jobscheduling	6
2.2 Other Asymmetric Systems	6
2.3 Linux	7
2.4 Current OS Design Techniques for SMT processors	9
2.4.1 Busy Waiting	9
2.4.2 Idle Loops	10
2.4.3 Intelligent Scheduling	11
2.5 Experimental System Details	11
3 Solution	13
3.1 Asymmetric Task Scheduling	13
3.2 Implementation	17
3.2.1 Linux Scheduler Modifications	17
3.2.2 Changes to the Linux SoftIRQ System	18
3.2.3 Control Module	19
4 Results	20
4.1 Basic Test Methodology	20
4.2 Homogeneous Workload Tests	21
4.3 Hybrid Tests	25
4.4 Odd Thread Count Behavior	27

5	Conclusions	29
5.1	Future Work	30
5.1.1	Hybrid SMP-SMT and Wider SMT CPUs	30
5.1.2	Fine-Grain Mode Adjustments	30
5.1.3	The MP scheduler	31
5.1.4	Dynamic Adjustments	31
A	Data	32
	Bibliography	39

List of Figures

2.1	Vertical and horizontal waste diagram	5
3.1	Modified scheduling algorithm for DK mode	14
3.2	Pipeline Modes	15
3.3	Actual scheduler modifications	18
4.1	Multi-Thread execution algorithm	22
4.2	Improvement of SMT and DK modes over UP for 4 threads (with network activity)	23
4.3	All projected hybrid run times	27
A.1	Art Data	33
A.2	Crafty Data	34
A.3	VPR Data	35
A.4	Vortex Data	36
A.5	Equake Data	37
A.6	Mesa Data	38

List of Tables

4.1	Categorization of benchmarks	22
4.2	Summary of data on Art	24
4.3	Summary of data on Mesa	24
4.4	Equake and Crafty hybrid data (4T)	26
4.5	Art and Mesa hybrid data (4T)	26

Chapter 1

Introduction

1.1 Motivation

Simultaneous multithreading (SMT) is a technique used to increase utilization of modern wide-issue superscalar processors [23, 24]. The focus of previous research into increasing SMT performance has been to treat the processor as a special case of a symmetric multiprocessing (SMP) system [13, 14, 16]. This is a natural starting point because a single SMT CPU is presented to the operating system as multiple independent processors (arranged in an SMP configuration). Thus, the current strategy for operating system design is to make an SMP-capable operating system aware of the special characteristics of the SMT, in an attempt to make more informed scheduling decisions [18]. Additional ways to improve performance exist that allow the operating system to intelligently schedule tasks in such a way that concurrent processes impact each other only slightly, or even possibly improve their collective performance [21, 20].

There are applications, however, that consistently exhibit lower performance when co-scheduled with another process on an SMT processor, compared to being scheduled by itself on a uniprocessor. These applications are able to effectively utilize most of the resources that would be shared in an SMT. For example, the most popular SMT implementation available provides only two floating-point functional units, both of which could be easily utilized by a single application.

When symmetric multiprocessing machines became available, software designers had to adjust their methods to take full advantage of the new hardware, as well as rethink previous performance-improving strategies that no longer applied to multiprocessors. As McDowell, et al. suggest [16], simultaneous multithreading represents a similar change, moving the target into a gray area, positioned somewhere between a uniprocessor and a multiprocessor. Exploration of possible fundamental software changes is necessary to achieve the maximum benefit from this new architecture, and therefore motivates our work.

A change to the design of a traditional multitasking operating system is provided here, which can offer additional ways for the system to utilize the unique characteristics of an SMT processor. By restricting asynchronous kernel activity to one set of machine contexts and user tasks to another, we can maximize performance for applications that efficiently utilize resources. Since kernel tasks are small and infrequently scheduled, the user application enjoys use of all resources for most of its execution time.

A major portion of our work is focused on a particular scenario. This is the case where a system utilizing an SMT processor runs in an environment where excessive network activity is possible, if not desired. Traditional Denial of Service (DoS) attacks are designed in a way that the target system is overwhelmed with requests, which are serviced (or at least attempted) by the operating system at the expense of all running application tasks. If the primary role of the system is to support a user application, then an excessive amount of network activity may cause starvation and a failure to meet a deadline. However, if the operating system had the ability, as we described above, to limit which types of activity can be processed by a certain CPU context, then starvation could not occur. The SMT processor provides a unique opportunity to divide a physical CPU in such a way that a certain Quality of Service (QoS) is achieved. This can neither be provided by a uniprocessor, nor a multiprocessor.

1.2 Contributions of this Thesis

The main contributions of this work are as follows. It:

- Describes a model for restricting tasks to a given context,
- Provides a modified Linux 2.6 kernel that implements our described model for a more flexible SMT scheduler,

- Provides a control module allowing runtime adjustments to the scheduler from userspace,
- Illustrates the advantages of our system through the presentation of performance metrics of known benchmark programs,
- Describes a possible system that automatically detects and transitions to an optimal operating mode, and
- Describes future research that could result from our work.

1.3 Organization of this Thesis

Chapter 2 describes the SMT architecture, specifically the hardware used to conduct our tests, and describes previous related research leading to our work. Chapter 3 describes the proposed changes to a conventional operating system scheduler and explains why these modifications lead to a more flexible system for achieving higher performance on SMT processors. Chapter 4 presents our test methodology used to evaluate our system, and the results we gathered. Chapter 5 reflects on the conclusions we draw from our work, and future research that could be performed to take our ideas further.

Chapter 2

Background

Applications rarely fully utilize modern processors. This is often due to the latency of other devices within a system, upon which the executing application depends. For in-order pipelined architectures, multithreading provides increased processor utilization by multiplexing the available resources among multiple instruction streams. The result is not a faster execution time for a single application, but rather an increased total throughput for multiple applications. Early multithreaded architectures, such as the MIT Alewife machine [4] and the Tera computer [6] provided fast hardware context switching, which allowed the pipeline to be loaded with an instruction from a different stream every cycle without penalty. Interactions between the running threads were minimal because only one instruction occupied a given pipeline stage at a time.

The introduction of superscalar architectures brought concurrent execution of multiple instructions from a single stream. Sufficient instruction-level parallelism (ILP) identified at runtime by the CPU increases throughput by processing multiple instructions per cycle in a single stage. Such an architecture is limited by the finite amount of ILP available in any instruction stream. Tullsen et al. describe this problem in terms of vertical and horizontal waste [23]. They diagram the execution stage of a wide superscalar pipeline vertically, as shown in Figure 2.1.

Simultaneous multithreading (SMT) is a technique used to increase resource utilization on modern wide-issue superscalar processors. When equipped with additional *hardware contexts* and SMT logic, a superscalar processor gains the ability to fetch instructions

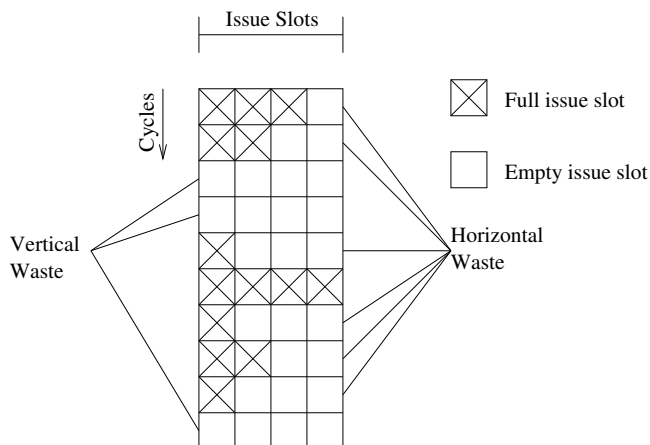


Figure 2.1: Vertical and horizontal waste diagram

from multiple streams (the number of streams equaling the total number of contexts in the processor). This allows the CPU to fill wasted issue slots with instructions from the other streams.

Vertical waste occurs when the CPU issues no instructions for a given cycle. Horizontal waste occurs when the CPU is unable to fill all available issue slots in a cycle, which is a result of low thread ILP. Simultaneous Multithreading focuses specifically on reducing horizontal waste by giving the CPU additional instruction streams from which it can fill the remaining issue slots. This can, however, simultaneously combat vertical and horizontal waste: a given cycle may be filled with instructions from any one of the current threads, or it may be filled with a combination of instructions from more than one thread.

SMT is a necessary improvement over the Tera design, when superscalar processors are in use. Tullsen et al. show that a fine-grained multithreading system capable of switching contexts each cycle without penalty is still unable to fill more than 40% of a wide superscalar's issue slots, even with a large number of instruction streams available. SMT, therefore, represents a unique solution to the problem of empty issue slots in modern superscalar architectures.

2.1 Symbiotic Jobscheduling

A simultaneous multithreaded processor introduces many complexities and unique characteristics over a similarly-equipped single-threaded processor. Most conventional multiprocessor systems replicate entire CPUs to achieve parallelism. Such a system requires an operating system to intelligently schedule processes to optimize cache performance and reduce false sharing. Typically, an operating system would ignore things such as the functional unit usage profile of concurrently executing applications, as the independent processors do not share these resources. These factors are exactly reversed in the case of an SMT processor, and provide the justification for specific operating system support for SMT CPUs. When co-scheduling two tasks on the virtual CPUs of an SMT, false sharing can actually increase performance (since the contexts share a cache) and processes need not be rescheduled on the same CPU, as the shared cache is warm no matter which context executes the process. Similarly, functional units are shared, and thus the operating system could increase performance by intelligently scheduling processes that would minimize interference.

Snavely et al. use the term *symbiosis* to describe the effectiveness with which multiple jobs achieve speedup when co-scheduled on a simultaneous multithreaded processor [19]. They describe a system, called SOS, that dynamically determines the most beneficial schedule for a set of tasks, based on their measured performances. This is referred to as *symbiotic jobscheduling*[21], and is a crucial step in improving performance on an SMT system. They take their idea further [20] by introducing weights into their system, providing the (very important) ability to support QoS within the system. This is a key concept, as it is similar to what we plan to do with our work, although it assumes that the operating system is always scheduled with the highest priority. Also, the SOS system spends execution time measuring the performance of many combinations of running tasks before determining the best schedule. Another solution, however, could be to classify applications statically, thereby eliminating the need for run-time sampling and preventing short-lived processes from hurting the performance of the system.

2.2 Other Asymmetric Systems

The idea of a single system with multiple processors dedicated to separate duties has been in practice since at least 1964. The Control Data Corporation (CDC) 6000 series

computers used this technique to allow I/O parallelism[8]. One or two central processors were used to perform high-speed arithmetic operations and general program execution, much like the CPUs of today's microcomputers. Additionally, small peripheral processors (PP) provided I/O service between external devices and central memory (CM). The PPs each had small private memories and access to all locations of the CM. By loading a small program into a PP's private memory, a complex I/O operation could be carried out while allowing the central processor to continue execution of other tasks. Thus, the CDC 6000 machine had multiple processors dedicated to different tasks executing different code. Much of the CDC operating system was implemented in PP code,¹ resulting in a system where most of the OS was executed on dedicated processors, minimizing interference with user tasks.

A more modern example of an asymmetric architecture is the IBM BlueGene/L machine (BGL)[22]. The BGL system packages two PowerPC 440 CPUs per processing element (PE). Unlike the CDC 6000 system (in which the CPUs were not similarly-equipped), a BGL PE has two equally-powerful processors in a pseudo-SMP configuration.² Although the system is capable of symmetric task assignment to both processors on a PE, normal operation utilizes one CPU for computation and the other for messaging. The BGL machine is an example of an SMP system where tasks are not symmetrically assigned. Effectively, the second physical processor becomes a service or co-processor, similar to the intent of our work with respect to an additional context of an SMT processor.

2.3 Linux

Linux is a modern multitasking operating system capable of operating on uniprocessor and multiprocessor systems. Time is divided coarsely into *epochs*, within which each task is given a finite amount of time to execute, called its *timeslice*. An epoch ends when all runnable tasks have exhausted their timeslice, thus ensuring that all tasks are given a chance to run. Tasks can be assigned priorities, which increase or decrease the timeslice that will be allotted to them each epoch. Each physical processor in the system is given two dedicated *runqueue* data structures, one of which holds tasks with time remaining this

¹The PPs used a different ISA from the central processor, and, therefore, required code to be specially written.

²The PowerPC 440 lacks actual SMP hardware, so the processors on a PE are not L1 cache-coherent.

epoch (runnable tasks), the other holds tasks who have exhausted their time (expired tasks). At the end of an epoch, the concrete runqueues switch roles; the expired runqueue holds all tasks, and, therefore, becomes the runnable queue for the next epoch.

The kernel enforces this strict timing strategy through the use of preemptive scheduling. The system timer is set by the kernel at boot to call a special routine every millisecond. Thus, during the execution of a task, the kernel regains control of the CPU periodically, where it can update timing measurements and choose to switch to another task if appropriate. Additionally, applications invoking I/O routines make calls directly into the kernel, and, thus, provide another entry point to the scheduler.

The main purpose of the task scheduler is to handle the execution of application tasks; these form the actual workload that the user of the machine intends to complete. The kernel also has work that it must perform, independent of any specific user task. Examples are flushing recent buffer cache changes to disk and balancing the task load on a multiprocessor system. The kernel wraps the routines to accomplish these duties in special kernel tasks, and inserts them into the runqueues as it would for a standard user task. Thus, these kernel tasks are given the same benefits as user tasks, such as guaranteed runtime and periodic scheduling.

Hardware interrupt handlers are divided into two pieces called the top and bottom halves. Although their names imply symmetry, in practice, the top half is much smaller than the bottom half [15]. The idea is that hardware interrupts should preempt the currently running task briefly, perform only necessary work, and then allow the preempted task to continue executing. This typically involves an acknowledgment to the hardware, and possibly the initiation of a data transfer to or from a buffer. The bottom half is typically executed later and performs the bulk of the real work necessary to handle the interrupt, such as copying data into userspace, or responding to another machine over the network. This is accomplished by marking a kernel task (`ksoftirqd`) as runnable, which will be scheduled in the normal fashion and will call the appropriate interrupt handler. There are two exceptions to this rule: first, an idle system will execute the bottom half immediately. Second, a heavily-loaded system may choose to force the execution of a bottom half if it is necessary to prevent buffer overflows.

When operating on a uniprocessor, the above scheme is quite straightforward. All preemptions due to kernel timers and hardware interrupts stop the entire system, and the kernel easily decides what to do next. A multiprocessor system, however, requires addi-

tional consideration for shared data structures as routines can be executed concurrently. When booting, only processor 0 is executing code, as instructed by the BIOS of the machine. This processor begins executing the early portions of the kernel and then prepares additional processors for execution. At some point, processor 0 sends an Inter-Processor Interrupt (IPI) to another processor (>0) that causes that CPU to begin executing code at a predefined point. At this time, multiple CPUs are executing the same kernel routines, but are choosing different tasks from their dedicated runqueue. Synchronization is maintained through extensive use of fine-grained locks to prevent concurrent modification of shared data structures. The considerations made by Linux for execution on SMT processors will be discussed in subsequent sections.

2.4 Current OS Design Techniques for SMT processors

The existing operating system we chose to modify already contains several improvements and considerations for execution on an SMT processor. These techniques, however, all assume the system will be used in a symmetric fashion, scheduling user and kernel tasks on either context. This behavior is a natural first choice for adding SMT processor support to an existing SMP-capable operating system. By treating the SMT CPU as a special case of an SMP, very few changes are required to utilize the additional context. This approach, however popular, can lead to lower performance for certain workloads. The most common operating system considerations for an SMT architecture are detailed in the following sections.

2.4.1 Busy Waiting

Traditionally, operating systems executing on symmetric multiprocessors make extensive use of busy waiting. If a process executing on one physical processor is waiting to access an object protected by a lock held by a process on another physical processor, a decision must be made about how to wait for the lock to be relinquished. If the process waiting has a significant amount of its timeslice remaining, the operating system should busy loop, waiting for the lock to become available. Since the execution time is allocated to the waiting process, the wait can pay off if the lock is released and useful work can be performed

before incurring the overhead of a context switch. If, however, the remaining timeslice is small (or the system is able to determine that the lock will be held for a significant amount of time), the context switch overhead can be justified in favor of allowing other tasks to use the CPU.

However, when the same scenario is applied to an SMT processor, the code making the decision must weigh other factors. Most importantly, busy looping in the waiting process will consume resources that the running process must use in order to complete its task and, in turn, release the lock. Therefore, busy looping can reduce performance instead of lowering latency as is the case in the multiprocessor. In the SMT scenario, the operating system may decide to use a waiting technique that does not impact the running process as heavily as would busy looping. The Intel architecture provides several methods for doing this, depending on the desired delay time.

2.4.2 Idle Loops

A multiprocessor-capable operating system is responsible for scheduling multiple processes on the available multiple processors. If the number of processors is greater than the number of runnable processes, then an idle loop is executed. Typically, this involves a loop that repeatedly checks for runnable tasks followed by an optional delay. On an SMP system, this provides the best performance by reducing the latency between a new process becoming runnable and the idling processor selecting it for execution. When all processor resources are independent, there is no reason not to have a waiting processor spin in a tight loop looking for newly-runnable processes.

If such a tight idle loop is used in the presence of an SMT, the idling context can reduce the performance of the running context, since it is consuming execution resources to constantly check the runqueues. The Intel architecture uses dynamic binding of resources to threads, which allows the CPU to effectively reduce to its non-multithreaded equivalent if one context is halted [17, 3]. The operating system, if equipped, can use this technique to halt the idling context for periods of inactivity, thereby allowing the running context full access to all available processor resources. Modern Linux 2.6 kernels employ this technique to improve performance under uneven and changing workloads.

2.4.3 Intelligent Scheduling

Typical multiprocessor operating systems are careful to reschedule tasks on the same CPU that they were previously running on. This increases performance in cases where the cache is still warm. If there is a chance that some data that the process will access is still in a processor’s cache, then it is most advantageous to run it there, eliminating compulsory misses after each context switch. An operating system that uses this technique may actually decide to not execute a runnable task on an available CPU because the cold-cache penalty will be greater than the wait for the warm-cache CPU. On a single physical SMT CPU, this technique is not necessary and may decrease performance.

Modern Linux 2.6 kernels are aware of this fact and ensure that logical CPUs that compose the same physical CPU are not biased against each other, therefore allowing a task to be rescheduled on either of the contexts of the warm-cache CPU. This is accomplished by assigning the contexts of a single physical CPU to the same runqueue. Thus, both contexts attempt to fetch new runnable tasks from the same location, eliminating bias between the two logical CPUs.

2.5 Experimental System Details

As of this writing, only two commercially-available processors incorporate simultaneous multithreading into their architectures: the Intel Pentium 4, and the IBM POWER5. Due to availability, popularity, and price, we use the Intel system as our experimental machine.

Some versions of the Pentium 4 include “HyperThreading Technology” (HT) support, which is the trademark name for Intel’s SMT support. The processor implements a 20-stage pipeline capable of issuing and retiring 3 instructions per cycle [17]. The CPU stores two architectural contexts allowing it to fetch from two independent instructions streams [9] each cycle.

The SMT implementation in the Intel Pentium 4 is considerably limited, compared to the designs used in most previous research. In other research, a Compaq Alpha processor³ was modeled [12], which was capable of fetching 8 instructions from 4 contexts per cycle. Such a system is capable of providing much larger performance gains as well as many more

³Note that the SMT Alpha was planned, but never released

execution options when compared to a 3-issue, 2 context machine. However, the Pentium is the only implementation available for our testing.

Chapter 3

Solution

In this chapter, we describe our proposed modifications to a conventional operating system scheduler. We used the Linux 2.6 kernel as the base for our actual modifications. Thus, we also describe the design of the related kernel structures and algorithms. Our changes to Linux are made primarily in the actual kernel source code. Other additions are mostly support-related (for example, the control interface) and are encapsulated within a loadable kernel module.

3.1 Asymmetric Task Scheduling

The Linux task scheduler is effectively unbiased between kernel and user tasks. Hence, tasks can be scheduled on either context of an SMT CPU. Asymmetric task scheduling (AST) refers to kernel tasks and user tasks being assigned to separate contexts. This way, the processor can remain mostly unpartitioned for fast execution of the user task (in cases where this is favorable) and then briefly enter partitioned mode when there are kernel tasks to execute. We call this the Dedicated Kernel (DK) mode of execution, as opposed to uniprocessor (UP) mode, where the CPU is treated as a normal uniprocessor, or simultaneous multithreading (SMT) mode, where the operating system treats the processor as a conventional SMP (with some consideration for it actually being an SMT). This assignment of tasks to contexts in DK mode is easily achieved by the algorithm shown in Figure 3.1.

```

schedule_next_task() {
    BEGIN:
    /* Existing performance tweak */
    if(next->allowed_cpus does not match this_cpu) {
        next = get_next_task();
        goto BEGIN
    }

    /* New assignment code */
    if(task->type matches this_cpu_duty) {
        goto RUN_TASK;
    } else {
        next->allowed_cpus -= this_cpu;
        next = get_next_task();
        goto BEGIN
    }
    RUN_TASK:
}

```

Figure 3.1: Modified scheduling algorithm for DK mode

Note that this is the basic algorithm present in Linux, with small modifications.

The first conditional block is an existing mechanism for providing CPU affinity in an SMP system. The `allowed_cpus` field of the task structure is a bit-mask; a set n^{th} bit indicates that processor n can execute the task. The second conditional block is novel and unique to our design. It checks if the next task selected for execution is one that is allowed to execute on the current context.

In practice, the new code is only executed a small number of times while the system adjusts the task list appropriately. As tasks transition from a blocked to runnable state in DK Mode, they are marked for execution only on the correct context. Thus, after every process in the system has been examined, each is marked to run on the correct context. Since both contexts fetch from the same runqueue, tasks not allowed to execute on the context looking for a new task are quickly identified at the top of the routine without re-executing the logic detailed in Figure 3.1.

Because this modification only slightly changes the existing task selection logic, it

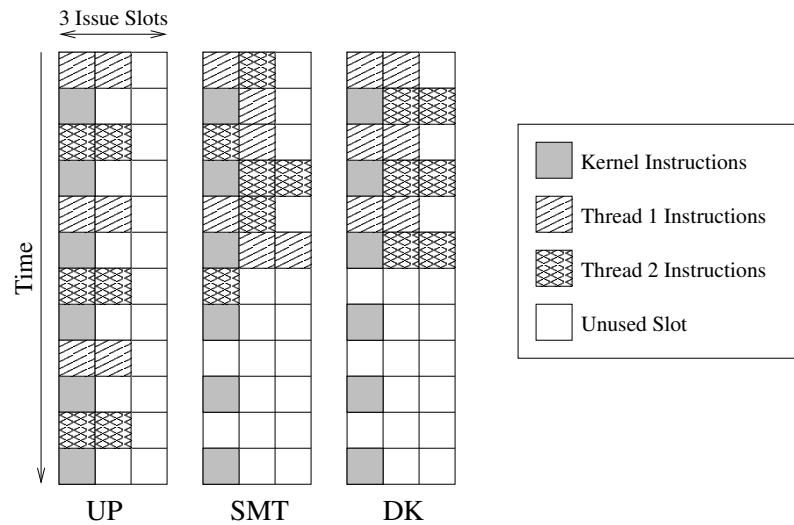


Figure 3.2: Pipeline Modes

is simple to selectively enable or disable this code at runtime allowing the operating system (or user) to select the mode that provides the best performance. A complete solution would encompass another block of code to reset the processor mask for each task, returning the system to its original state. This could easily be accomplished by saving the original `allowed_cpus` mask when DK mode is enabled, so that it can be restored when DK mode is disabled.

In Linux, when there are fewer runnable processes than available processors, a special kernel task, called the idle task, is scheduled on unused CPUs. The idle task is always available for running and is chosen only if there are no real tasks available. When running on an SMT, the idle task explicitly issues a halt instruction, which delays execution until the next hardware interrupt. On the Intel architecture, a halt issued to one context of an SMT CPU relinquishes all shared resources (i.e., unpartitions) to the other running context. Therefore, when only one task is runnable, the Linux idle task collapses the unused context, which generally increases the performance of the running context. The goal of our proposed change is to prevent one context from running user tasks. By preventing it from “seeing” any available user tasks it is collapsed when there are no runnable kernel tasks.

Figure 3.2 shows an example of a specific workload slice, as processed in UP, SMT, and DK modes. The user application threads are being executed with some periodic kernel work (such as a network data transfer). The execution stage of the pipeline is diagrammed

vertically with time and is replicated for each mode. The pipeline is 3-wide, and has only two floating-point (FP) units, just like our Pentium-4 test machine. This means that at most two FP instructions can be processed per cycle. Additionally, the instructions must be processed in the same order in all three examples, as we assume dependencies have been resolved. This example is a simplified representation of what actually happens at runtime, as it ignores things like context-switching overhead; it is, however, sufficient to convey the idea.

In the UP case, two FP instructions from each alternating thread are executed per cycle while handling the kernel work in between. The multiprogrammed operating system scheduler strictly alternates between the user threads. In the SMT case, the scheduler has activated the second context of the machine and is attempting to issue instructions from each stream simultaneously. Thus, two floating point instructions are executed per cycle, one from each stream. Since the kernel activity takes the place of one of the contexts at time 2 and thread 1 has only one more FP instruction that can be executed (according to dependency rules), only one FP instruction is executed that cycle. The DK example does not attempt to fetch instructions from multiple user streams, and, thus, executes much like the UP case. The difference, however, is that the user tasks need not pause for the kernel activity and can be processed simultaneously, because they use different functional units. The result is that both tasks finish one cycle ahead of their SMT counterparts and five cycles ahead of UP.

This example demonstrates a situation where a scheduling policy based on our model can help increase instruction throughput. We used a floating-point example because it introduces additional constraints that are useful for illustrating possible bottlenecks. In practice, some integer-based applications also benefit from executing in DK mode. Similarly, not all floating-point applications exhibit the behavior described above.

A possible variation of our model could be used in the case where DK mode is almost always favorable for the intended workload. In this case, separating the runqueues into kernel-only and application-only queues, and properly assigning them to contexts could help reduce the overhead incurred by the scheduler when the next task on top of the shared runqueue is avoided. However, when operating temporarily in normal SMT mode, such a kernel might suffer from a less effective cache strategy, as was previously mentioned in reference to typical SMP configurations.

Kernel code is comprised of integer-based work, but performs worse than a typical

integer application. Previous studies have shown that the operating system executes so infrequently that branch predictions and cache performance suffer [18, 11, 5]. This behavior results in favorable conditions for user applications co-scheduled in the adjacent context. First, the operating system is interleaved with application execution. Hence, poor performance in kernel routines will not delay the execution of the application. Second, the dynamic nature of resource binding in the Intel CPU causes the poorly-performing kernel code to utilize few resources, thus minimizing the impact of the concurrently executing user task.

We propose that the interrupt-handling mechanism in Linux be altered slightly as well, in a way similar to the scheduler. When running in DK mode, bottom halves are only ever executed on the context assigned to kernel tasks. This allows the system to execute the interrupt bottom halves concurrently with a running user task without ever interrupting it. In practice, this may occur naturally, but in a situation where interrupt flooding is present (for example, a DoS attack), it may be desirable to guarantee that starvation does not occur for the user task.

3.2 Implementation

In order to evaluate our proposed operating system changes, we chose to modify the Linux 2.6.8.1 kernel. Before our modifications, this kernel contained scheduler optimizations for a single runqueue per physical processor as well as an optimized idle loop for SMT processors.

3.2.1 Linux Scheduler Modifications

The changes we made to the Linux scheduler are almost identical to those proposed in Figure 3.1. At the last moment before the scheduler prepares the next task for execution, the code segment depicted in Figure 3.3 is run.

The first condition causes the block to be entered only if DK mode is enabled and the current CPU is the one dedicated to only kernel tasks. The second condition is true if the task that the scheduler just selected for execution next is a user task. In this case, the CPU mask is set on the user task so that it will not be considered for later execution

```

if(DK_ISON && DK_THISCPU) {
    if(next->mm != NULL) {
        cpu_clear(cpu, next->cpus_allowed);
        next = rq->idle;
        goto switch_tasks;
    }
}

```

Figure 3.3: Actual scheduler modifications

on this CPU. The last two lines select the idle task for execution and jump directly to the context switching code. This is a standard way to select an alternate task as the idle code immediately checks for runnable processes before idling. This ensures that the selection code (before our code shown in Figure 3.3) for another runnable task is properly re-run.

3.2.2 Changes to the Linux SoftIRQ System

As previously mentioned, the top half interrupt processing system is left unchanged. This simplifies the modifications we need to make to the system as the existing device drivers and other hardware-related code depend on this system behaving in a certain way. Due to the dual-phase nature of the interrupt system, this is of little consequence to our performance measurements as the top halves of interrupts are designed to run for a very short amount of time, minimizing their impact on the preempted task. Also, the latency of many parts of the system may depend on the speed at which the top halves execute. Thus, we are able to keep latency low by allowing the responses to the hardware to execute as they do in a normal system. We did, however, modify the behavior of the bottom half interrupt processing system.

The primary method for handling interrupt bottom half processing in Linux is through the use of *softirqs*. When, for example, a device driver registers its top half Interrupt Service Routine (ISR) with the operating system, it also registers its bottom half with the *softirq* system. This assigns the bottom half a unique integer index, which can be used by the top half to notify the system that its corresponding bottom half should be scheduled for execution soon. When the *softirq* is raised, it is assigned to be processed by one of the

processors in the system. We still allow this to happen in our modified kernel but alter the resulting behavior by changing the logic in the *softirq* processing routine (`_do_softirq()`).

Both processors routinely execute the `_do_softirq()` routine. Invocation can occur from many different locations scattered in the code; instead of identifying every call, we simply modified the routine itself to exhibit the behavior we desire. As such, when DK mode is enabled, the context dedicated to user tasks only runs *softirqs* that match a mask set dynamically at runtime. This is needed because some kernel tasks must be run by both contexts in order to keep the system running. However, the majority of the *softirqs* are masked out of the user context. The kernel context always processes all pending *softirqs*. Additionally, it collects *softirqs* that were previously scheduled for execution on the user context. In order to handle this safely, locking was added to the code for raising and clearing *softirqs*.

Linux also provides another mechanism for sequential queuing and execution of lower-priority bottom halves: *tasklets*. The tasklet mechanism simply chains scheduled work together and processes it sequentially, instead of the priority-based *softirq* system. In fact, tasklet processing is invoked from the lowest-priority *softirq*. Similar modifications were made to the tasklet system to ensure that all were processed by the dedicated kernel context.

3.2.3 Control Module

The control module is relatively simple. Its primary purpose is to provide a way to transition from normal SMT mode to DK mode as well as adjust the *softirq* mask for the user context. Inserting the module immediately transitions to DK mode, and a *proc interface* is provided for adjusting the mask. The module also provides some diagnostic information, which we use to verify that our modifications to the interrupt system are having the desired effects. Loading the module into the kernel at runtime effectively enables the switching between SMT and DK mode from userspace.

Chapter 4

Results

4.1 Basic Test Methodology

We evaluated our operating system changes by measuring the performances of SPEC2000 benchmarks[2, 1] using our mode of operation compared to conventional modes. Our modification has little to offer without additional kernel tasks to process in the adjacent context. Thus, we need a method for introducing an increased amount of kernel activity to the system. We chose to generate load on the interrupt system and did so by designing a simple application pair that creates client-server network traffic. The machine under test ran a benchmark program while also acting the server role to a standalone client on another machine.

Our client-server application implements a protocol similar to many common ones, where large blocks of data are continually being requested by the client and supplied by the server. This activity resembles the I/O-bound load that a simple web server or custom data feedback application would generate. A small amount of occasional client-to-server traffic combined with a large amount of server-to-client traffic resembles a real-life situation.

In all tests involving the execution of SPEC benchmarks, we carefully ensured that the desired number of threads were running at all times. For a four-thread test, four instances of the application under examination were launched simultaneously. A small control application was written that used the algorithm in Figure 4.1 to ensure that all threads

run with the same resource contention for the duration of the measured iteration. The structure of this algorithm lets us run a certain number of threads of the same application, automatically restarting a thread if it finishes so that the remaining threads continue to run at the same speed. The explicit `sched_yield()` call ensures that the new task forfeits its right to run immediately allowing all threads a chance to be forked before the first one begins its task. The timing data for the restarted threads is not used in calculations, only to ensure deterministic test results. This technique is crucial for consistent results; if three of the four threads finish early, the remainder of the last thread must be executed with the same resource contention. Otherwise, the performance could be different than that of the other threads.

We chose to eliminate the case of executing a single application thread for obvious reasons. As previously described, Linux will collapse a context of the SMT when there is no runnable task to execute. Therefore, a typical SMT implementation with only a single thread to execute will naturally behave much like our enhanced kernel because there is already an available context waiting to handle any additional kernel activity.

The value used to compare the performances of two scenarios is time-to-completion (TTC). This metric is the time from when the threads are forked to the time that the slowest thread finishes its first iteration. Average iteration times were also measured but did not differ significantly (except in a specific case, which will be discussed later). We chose to report the TTC value, as it gives a more realistic picture of the performance that could be expected from a multi-threaded application where a single iteration can only be considered complete after all threads finish their work. Additionally, we found circumstances where the complex nature of the interactions between threads can vary the individual iteration times, but the TTC value remains relatively constant.

4.2 Homogeneous Workload Tests

The most basic test scenario we evaluated was the case of a single application run in two, three, or four concurrent threads. The timing results for all three kernel modes (UP, SMT, and DK) were compared when running this configuration with and without the presence of network activity, as created by our client-server test application. The benchmarks chosen were a mix of floating-point and integer programs that would fit completely into a

```

fork n times
if(parent) {
    while(number_complete < n) {
        wait_for_child_exit()
        respawn a new child
    }
} else {
    sched_yield()
    exec SPEC benchmark
}

```

Figure 4.1: Multi-Thread execution algorithm

Category	Member Benchmarks
pro-SMT	Vortex, Mesa, Equake
pro-DK	Vpr, Crafty, Art

Table 4.1: Categorization of benchmarks

quarter of our available memory without the need for paging.

Analysis of the data yielded by our tests confirms some hypotheses and raises new unanswered questions. First, DK mode does, in fact, behave almost identically to UP mode in the absence of network activity. Second, DK mode always outperforms UP mode in the presence of network activity. These facts, now confirmed, make it easy to classify the test benchmarks into two categories, which we will call pro-SMT and pro-DK. The naming of the latter is justified because DK always performs as well or better than UP, while exhibiting many of the same characteristics. The assignment to categories is shown in Table 4.1. This is interesting because, ignoring network activity, the pro-DK category is also the pro-UP category. Thus, applications that perform better on a uniprocessor than on an SMT also perform better even when network activity is present, in DK mode. Thus, adding the DK functionality to the operating system provides the ability to utilize the additional resources and capabilities provided by the SMT processor without running in a normal SMT mode when such a mode is detrimental to performance of a specific application.

A comparison of the gains realized by both SMT and DK modes over UP is given in Figure 4.2, in order of the best DK performance to least. That the first three applications

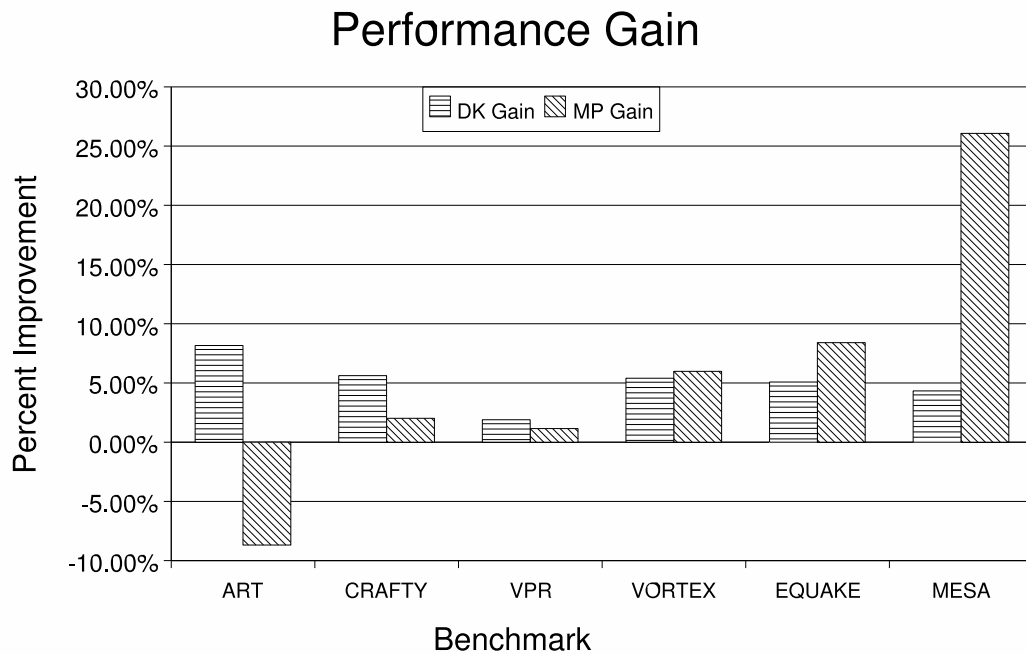


Figure 4.2: Improvement of SMT and DK modes over UP for 4 threads (with network activity)

see a larger benefit when running in DK mode compared to SMT mode. The last three applications, however, benefit from SMT mode. This leads us to believe that applications in the pro-SMT category always benefit from being co-scheduled in a normal multiprocessing manner. Additionally, applications that perform the same or worse on a traditional SMT system than on a uniprocessor, see a benefit by moving to DK mode.

The data for Art and Mesa in all modes and all thread configurations is shown in Tables 4.2 and 4.3. These applications represent our best and worse cases, respectively. The three-thread SMT performance of both applications is anomalous, and is discussed in Section 4.4.

Art’s runtime is longer when running in SMT mode than in UP mode, which immediately shows that it is best run on a conventional uniprocessor. Given this fact, it is not surprising that, in the presence of network load, that it performs better in DK mode than UP. This is because the CPU is able to handle the network load using the additional context of the CPU instead of competing for time, as in the UP case.

Mesa’s behavior is almost the opposite of Art. While DK mode does not hurt

Mode	Config	Without Net		With net	
		TTC	%-Change	TTC	%-Change
UP	2T	84.67	0.00%	95.67	0.00%
	3T	129.33	0.00%	146.33	0.00%
	4T	170.00	0.00%	192.00	0.00%
SMT	2T	86.00	1.55%	103.33	7.41%
	3T	159.00	18.66%	179.00	18.25%
	4T	173.33	1.92%	207.67	7.55%
DK	2T	85.00	0.39%	88.00	-8.72%
	3T	125.33	-3.19%	135.33	-8.13%
	4T	170.33	0.19%	176.33	-8.89%

Table 4.2: Summary of data on Art

Mode	Config	Without Net		With Net	
		TTC	%-Change	TTC	%-Change
UP	2T	327.67	0.00%	378.00	0.00%
	3T	491.67	0.00%	576.67	0.00%
	4T	659.00	0.00%	777.33	0.00%
SMT	2T	438.67	25.30%	286.67	-24.16%
	3T	426.33	-15.33%	492.67	-14.57%
	4T	520.33	-26.65%	574.67	-26.07%
DK	2T	340.67	3.82%	367.67	-2.81%
	3T	518.33	5.14%	549.67	-4.91%
	4T	702.33	6.17%	743.67	-4.52%

Table 4.3: Summary of data on Mesa

performance for Mesa, SMT mode clearly wins. Mesa does not incur a performance hit by being co-scheduled on an SMT, which appears to be due to low instruction throughput and large memory demands. This is a case where the traditional SMT approach is appropriate and beneficial. If Mesa frequently executes latency-inducing instructions, there will be many issue slots available for the other context to utilize. In contrast, Art appears to have high throughput and is able to utilize most of the CPU resources in a single instruction stream.

4.3 Hybrid Tests

Our modifications to the operating system were made in such a way that transitioning between DK and normal SMT mode is easily accomplished. Knowing that DK mode will not always offer a performance benefit in every scenario and with every application, it is desirable to be able to select the mode that will provide the best performance. To test the performance benefit of running a hybrid SMT+DK kernel, we designed a dynamic testing scenario. By running two of our homogeneous tests back-to-back with a mode change in between, we were able to measure the performance of a heterogeneous code mix on all kernel modes and determine which is most efficient.

We planned to test this simple hybrid strategy by choosing two applications, one of which benefited more from SMT mode, another benefiting more from DK mode. Clearly, a fixed SMT or DK policy cannot offer better performance than hybrid mode, where each application is executed in its most favorable mode.

Because our DK-modified kernel is actually an SMT kernel, disabling DK mode causes it to perform exactly like an SMT. Thus, we can predict the performance of a hybrid run by simply combining the appropriate results from previous runs. We verified this fact, and thus provide examples of how to calculate the hybrid run times from our homogeneous test data. Since the test timings include overhead, such as that incurred by `fork()` and `exec()`, only the time needed to switch modes is left out of a simple addition. The work performed to switch modes is extremely light and can safely be left out of the prediction. Using the above assumptions, we can predict that a hybrid quake-crafty test can be determined by the following equation:

$$436.67 + 404.33 = 841.00$$

The hybrid mode allows one to select the fastest run of each application in the computation of the overall sum. By referencing the data for the independent runs of quake and crafty, as shown in Table 4.4, we observe that the overall runtime in hybrid mode is shorter than either of the other modes alone.

A combined run of Art and Mesa show a situation where hybrid mode could improve performance. As shown in Table 4.5, DK mode improves performance for Art significantly, but (like UP mode) Mesa’s performance is severely limited when compared to SMT mode. In this case, the time savings when running Mesa in SMT mode dominates

Mode	Crafty	Equake	Total
UP	462.67	441.33	904.00
SMT	453.33	404.33	857.66
DK	436.67	419.00	855.67
Hybrid	436.67	404.33	841.00

Table 4.4: Equake and Crafty hybrid data (4T)

Mode	Art	Mesa	Total
UP	192.00	777.33	965.33
SMT	208.67	574.67	783.34
DK	176.33	743.67	920.00
Hybrid	176.33	574.67	751.00

Table 4.5: Art and Mesa hybrid data (4T)

that of Art’s benefit with DK mode, which would be an indicator that a static SMT mode would be better than a static DK mode choice. A hybrid system eliminates the need to compromise. SMT mode alone offers a large 18.9% improvement over UP mode, but hybrid execution increases this gain to 22.2%.

A comparison of all projected hybrid run times is given in Figure 4.3. In some cases the difference between a hybrid and static mode is so small that the benefit may not justify the complication. In others, a significant improvement can be made. The figure shows that hybrid mode is always better than any other mode when running two applications that each favor a conflicting mode.

We have shown that when faced with execution of two applications that favor different modes, a hybrid approach can yield the best performance, compared with simply choosing the mode that will provide a performance increase for the one that is larger than the decrease incurred by the other application. The hybrid mode provides the performance increases for each application.

Although we recognize that few real-world workloads will be easily divided into long sections of work that require only infrequent mode changes, we present additional techniques in Chapter 5 which may account for mixed workloads and should yield similar results. We encourage the future development of these ideas to make our system applicable to a wider range of real-world tasks.

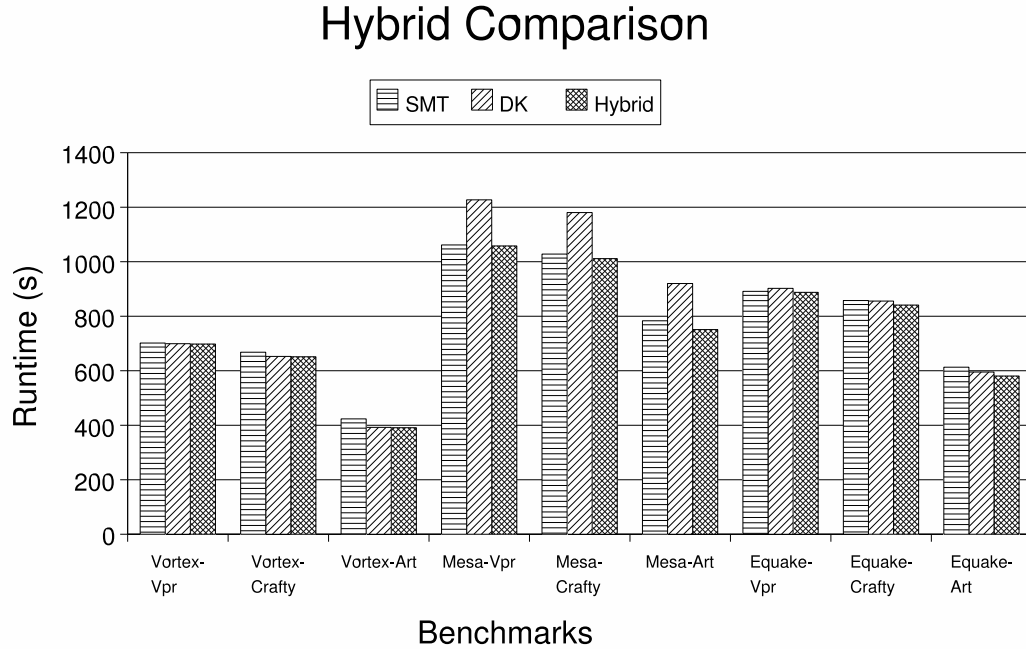


Figure 4.3: All projected hybrid run times

4.4 Odd Thread Count Behavior

In addition to the intended results we recorded from our tests, another unrelated conclusion became apparent. Our original experimentation methodology was to test configurations of two, three, and four threads for each mode. In doing so, we noticed a large discrepancy between the average iteration time and the TTC for tests running three threads in SMT mode. While the average time remained relatively constant, and was close in value to our expectations, the TTC varied and was always significantly greater than the average. Since neither of the other test sizes exhibited this behavior, we hypothesized that the strange effect may be due to inefficient scheduling of an odd number of threads. Experiments with a five-thread test confirmed this. Further examination revealed that, in fact, the standard Linux scheduler was not being completely fair to each of the three threads. It appeared that the threads were sequentially pinned to CPUs. In this case, an even number of threads (or, more accurately, a thread count which is a multiple of the number of CPUs in the system) resulted in an evenly distributed load. An odd number of threads resulted in

an uneven load. In such a case all processes should be scheduled round-robin on the available processors to ensure a fair time distribution. Clearly, the current scheduler is in need of further examination and, hopefully, an adjustment.

This unbalanced scheduling situation does not always appear, although it does in the majority of the cases. Table 4.2 shows the significant decrease in SMT performance in the three-thread configuration in both the presence and absence of network load. Crafty shows this same behavior in the network-absent three-thread case. However, the network-present case seems to perform closer to expectations. We believe that since the additional increased network load takes the form of a kernel task, it may help round out the scheduler's decisions, eliminating the unfair time allocation.

Chapter 5

Conclusions

In this thesis, we explored an alternative way to approach utilization of the additional resources provided by an SMT processor. We moved away from traditional symmetric scheduling techniques and suggested an operating system modification that would illuminate a new path in light of the diminishing returns resulting from conventional thinking. We achieved this goal and did so while maintaining compatibility with existing algorithms as well as the added benefit of being able to utilize both existing and new paradigms in the same system at runtime.

Our research produced:

- Experiments showing that some applications perform better when scheduled alone as opposed to being co-scheduled on an SMT processor,
- A categorization of the tested benchmarks into those which do and do not perform well on an SMT,
- The realization that poor SMT performers can achieve a performance gain through the use of our operating system modification,
- A technique for combining our modifications with traditional scheduling algorithms at runtime can that achieve a best-of-both-worlds scenario, and
- Shortcomings of the current Linux scheduling algorithm in the case of a non-multiple number of compute-bound threads.

5.1 Future Work

In the process of modifying the Linux kernel and examining the results of our tests, many new ideas were formed about ways to further improve performance. Additionally, we recognized situations where our system could not be directly applied to existing workloads and implementations. We attempt to provide possible solutions to increase the usability of the system in real-world situations. These ideas are presented here as the basis for future work.

5.1.1 Hybrid SMP-SMT and Wider SMT CPUs

The concept of a system with multiple SMT processors offers some interesting ways to apply our ideas. Certainly, our model could be adapted to allow independent control of the operating mode for each physical processor. This could provide additional gains for mixed workloads, especially if it is desirable to concurrently execute multiple applications with differing mode preferences. By maintaining a DK scheduling policy on one processor and a conventional SMT policy on another, three differing applications could simultaneously realize an increase in performance.

Similarly, an SMT processor with a larger number of contexts and a wider pipeline, such as the Alpha 21464 described in previous research, could offer more mode options [12]. We have shown that some applications benefit from being co-scheduled on an SMT compared to a uniprocessor. Perhaps this characteristic is specific to our machine, and would not apply to the same application on a wider SMT superscalar. In this case, the operating system may provide more control, such as collapsing two out of four contexts to allow the remaining two contexts to access more resources. The end result may be a software-controlled partitioning scheme where the operating system can choose to treat the CPU as a single-context wide superscalar, a two-context SMT, or a four-context narrow SMT.

5.1.2 Fine-Grain Mode Adjustments

In an effort similar to the above technique, fine-grained adjustments could be made during concurrent execution to allow mixed workloads to benefit from their preferred mode of execution, even with a single physical processor. The Linux scheduler has a concept of an *epoch*[7]. When an epoch starts, all processes are given a finite amount of CPU time called

a *timeslice*. An epoch ends when all processes have consumed their timeslice. In order to accommodate mode changes on a fine-grained level, this algorithm could be modified slightly. First, each epoch would be executed in either DK or SMT mode alternating each time. Tasks that are marked as *preferred-DK* or *preferred-SMT* would be given twice the timeslice for the epochs that run in their preferred mode. For the other epochs, they would be allotted no timeslice. Thus, each task would be executed in the appropriate mode. In a design such as this, latency would be increased; in the case of compute-bound tasks however, this would be of no consequence.

5.1.3 The MP scheduler

The idea of a mode-oriented operating system scheduler for SMT CPUs, called the MP scheduler, was originally presented by Freeh [10]. This idea aimed at improving SMT performance of applications that run best in a uniprocessor environment. The DK mode achieves part of these objectives. We have shown that some applications do (even in the absence of network activity) perform better when not co-scheduled in a typical SMT fashion. Work on the MP scheduler has not yet begun, but our work clearly demonstrates that there are cases where improvement is available. This idea, combined with either fine-grained mode adjustments or a true multiprocessor, could add significant value to a real system.

5.1.4 Dynamic Adjustments

Although we believe that an application fundamentally belongs to either the pro-SMT or pro-DK category (as described in section 4.2), it may be desirable to have the operating system capable of sampling performance indicators and making adjustments dynamically to maximize gain. This may be in a scenario where an application's categorization is unknown or where its categorization changes during execution. By making use of the performance counters present in the Pentium 4 CPU, the operating system could periodically examine performance and re-categorize applications at runtime. A system similar to this, called SOS, was described by Snively et al. [21].

Appendix A

Data

Test	No-NET		With-NET		Δ Runtime	Δ TTC
	Avg Runtime	TTC	Avg Runtime	TTC		
UP-2T	82.00	83.00	97.50	98.00		
UP-2T	82.50	84.00	93.00	94.00		
UP-2T	84.00	87.00	92.50	95.00		
2T Avg	82.83	84.67	94.33	95.67	12.19%	11.50%
UP-3T	123.67	128.00	142.67	144.00		
UP-3T	125.33	128.00	141.67	146.00		
UP-3T	123.67	132.00	144.67	149.00		
3T Avg	124.22	129.33	143.00	146.33	13.13%	11.62%
UP-4T	165.25	169.00	187.00	192.00		
UP-4T	162.50	168.00	186.50	191.00		
UP-4T	163.25	173.00	187.50	193.00		
4T Avg	163.67	170.00	187.00	192.00	12.48%	11.46%
MP-2T	87.00	87.00	94.00	107.00		
MP-2T	85.00	85.00	91.50	101.00		
MP-2T	86.00	86.00	93.50	102.00		
2T Avg	86.00	86.00	93.00	103.33	7.53%	16.77%
MP-3T	123.00	142.00	127.33	158.00		
MP-3T	129.67	167.00	129.67	174.00		
MP-3T	128.67	168.00	142.67	205.00		
3T Avg	127.11	159.00	133.22	179.00	4.59%	11.17%
MP-4T	171.25	172.00	181.75	200.00		
MP-4T	171.50	173.00	184.00	210.00		
MP-4T	173.00	175.00	186.50	216.00		
4T Avg	171.92	173.33	184.08	208.67	6.61%	16.93%
DK-2T	81.00	82.00	87.00	91.00		
DK-2T	84.00	86.00	85.50	87.00		
DK-2T	82.00	87.00	86.00	86.00		
2T Avg	82.33	85.00	86.17	88.00	4.45%	3.41%
DK-3T	120.67	123.00	132.33	133.00		
DK-3T	123.00	126.00	132.33	133.00		
DK-3T	120.33	127.00	133.67	140.00		
3T Avg	121.33	125.33	132.78	135.33	8.62%	7.39%
DK-4T	159.75	165.00	175.75	177.00		
DK-4T	164.75	173.00	174.00	175.00		
DK-4T	163.25	173.00	176.00	177.00		
4T Avg	162.58	170.33	175.25	176.33	7.23%	3.40%

Effect of adding an additional thread			
	UP	MP	DK
2T-3T	33.32%	32.34%	32.14%
3T-4T	24.10%	26.06%	25.37%

Benefit of DK (TTC)			
	2T	3T	4T
UP	-0.39%	3.09%	-0.20%
MP	1.16%	21.17%	1.73%
UP-NET	8.01%	7.52%	8.16%
MP-NET	14.84%	24.39%	15.50%

Figure A.1: Art Data

Test	No-NET		With-NET		Δ Runtime	Δ TTC
	Avg Runtime	TTC	Avg Runtime	TTC		
UP-2T	199.00	200.00	230.00	230.00		
UP-2T	198.00	199.00	229.50	230.00		
UP-2T	198.00	199.00	230.50	231.00		
2T Avg	198.33	199.33	230.00	230.33	13.77%	13.46%
UP-3T	298.00	299.00	344.67	346.00		
UP-3T	298.00	299.00	344.67	345.00		
UP-3T	298.67	300.00	344.00	345.00		
3T Avg	298.22	299.33	344.45	345.33	13.42%	13.32%
UP-4T	398.25	400.00	460.00	463.00		
UP-4T	398.50	401.00	460.75	463.00		
UP-4T	398.00	399.00	460.00	462.00		
4T Avg	398.25	400.00	460.25	462.67	13.47%	13.54%
MP-2T	199.50	200.00	213.50	227.00		
MP-2T	198.50	199.00	214.00	227.00		
MP-2T	199.00	199.00	212.50	213.00		
2T Avg	199.00	199.33	213.33	222.33	6.72%	10.34%
MP-3T	300.00	337.00	313.33	336.00		
MP-3T	290.33	327.00	314.67	338.00		
MP-3T	294.00	339.00	314.33	336.00		
3T Avg	294.78	334.33	314.11	336.67	6.15%	0.69%
MP-4T	399.00	400.00	429.75	454.00		
MP-4T	400.25	402.00	428.50	455.00		
MP-4T	400.75	402.00	428.75	451.00		
4T Avg	400.00	401.33	429.00	453.33	6.76%	11.47%
DK-2T	199.00	200.00	217.50	218.00		
DK-2T	201.00	202.00	216.00	217.00		
DK-2T	200.00	201.00	217.00	217.00		
2T Avg	200.00	201.00	216.83	217.33	7.76%	7.52%
DK-3T	299.33	300.00	326.00	327.00		
DK-3T	300.00	300.00	322.67	323.00		
DK-3T	301.67	303.00	327.67	330.00		
3T Avg	300.33	301.00	325.45	326.67	7.72%	7.86%
DK-4T	401.00	402.00	436.50	438.00		
DK-4T	399.00	399.00	434.50	436.00		
DK-4T	399.50	400.00	433.00	436.00		
4T Avg	399.83	400.33	434.67	436.67	8.01%	8.32%

Effect of adding an additional thread			
	UP	MP	DK
2T-3T	33.50%	32.49%	33.41%
3T-4T	25.12%	26.31%	24.89%

Benefit of DK (TTC)			
	2T	3T	4T
UP	-0.84%	-0.56%	-0.08%
MP	-0.84%	9.97%	0.25%
UP-NET	5.64%	5.41%	5.62%
MP-NET	2.25%	2.97%	3.68%

Figure A.2: Crafty Data

Test	No-NET		With-NET		Δ Runtime	Δ TTC
	Avg Runtime	TTC	Avg Runtime	TTC		
UP-2T	203.00	204.00	243.00	244.00		
UP-2T	204.00	204.00	241.00	242.00		
UP-2T	204.00	204.00	241.50	242.00		
2T Avg	203.67	204.00	241.83	242.67	15.78%	15.93%
UP-3T	307.00	307.00	364.00	365.00		
UP-3T	305.00	306.00	375.00	376.00		
UP-3T	306.00	307.00	374.00	375.00		
3T Avg	306.00	306.67	371.00	372.00	17.52%	17.56%
UP-4T	408.00	408.00	498.00	499.00		
UP-4T	408.00	408.00	487.00	488.00		
UP-4T	408.00	409.00	490.00	491.00		
4T Avg	408.00	408.33	491.67	492.67	17.02%	17.12%
MP-2T	204.00	205.00	220.50	236.00		
MP-2T	203.50	204.00	220.00	236.00		
MP-2T	204.00	204.00	221.00	237.00		
2T Avg	203.83	204.33	220.50	236.33	7.56%	13.54%
MP-3T	311.00	422.00	305.33	393.00		
MP-3T	306.33	415.00	337.00	444.00		
MP-3T	304.33	408.00	303.67	389.00		
3T Avg	307.22	415.00	315.33	408.67	2.57%	-1.55%
MP-4T	410.25	412.00	445.50	487.00		
MP-4T	411.75	417.00	448.00	492.00		
MP-4T	409.00	410.00	446.00	482.00		
4T Avg	410.33	413.00	446.50	487.00	8.10%	15.20%
DK-2T	217.00	218.00	245.00	246.00		
DK-2T	225.00	226.00	235.00	236.00		
DK-2T	214.50	219.00	231.50	233.00		
2T Avg	218.83	221.00	237.17	238.33	7.73%	7.27%
DK-3T	319.00	319.00	354.00	355.00		
DK-3T	331.00	331.00	355.00	355.00		
DK-3T	324.00	328.00	360.00	361.00		
3T Avg	324.67	326.00	356.33	357.00	8.89%	8.68%
DK-4T	431.00	431.00	480.25	481.00		
DK-4T	428.00	428.00	485.00	486.00		
DK-4T	452.00	452.00	476.25	483.00		
4T Avg	437.00	437.00	480.50	483.33	9.05%	9.59%

Effect of adding an additional thread			
	UP	MP	DK
2T-3T	33.44%	33.65%	32.60%
3T-4T	25.00%	25.13%	25.71%

Benefit of DK (TTC)			
	2T	3T	4T
UP	-8.33%	-6.30%	-7.02%
MP	-8.16%	21.45%	-5.81%
UP-NET	1.79%	4.03%	1.89%
MP-NET	-0.85%	12.64%	0.75%

Figure A.3: VPR Data

Test	No-NET		With-NET		Δ Runtime	Δ TTC
	Avg Runtime	TTC	Avg Runtime	TTC		
UP-2T	96.00	97.00	113.00	114.00		
UP-2T	96.00	97.00	112.50	114.00		
UP-2T	96.50	97.00	113.00	114.00		
2T Avg	96.17	97.00	112.83	114.00	14.77%	14.91%
UP-3T	143.67	146.00	169.67	172.00		
UP-3T	143.00	144.00	170.00	170.00		
UP-3T	143.67	144.00	170.00	171.00		
3T Avg	143.45	144.67	169.89	171.00	15.56%	15.40%
UP-4T	191.25	193.00	226.50	229.00		
UP-4T	192.25	193.00	226.25	228.00		
UP-4T	192.25	194.00	225.50	228.00		
4T Avg	191.92	193.33	226.08	228.33	15.11%	15.33%
MP-2T	97.00	97.00	102.00	108.00		
MP-2T	93.00	93.00	103.00	110.00		
MP-2T	94.00	95.00	101.00	108.00		
2T Avg	94.67	95.00	102.00	108.67	7.19%	12.58%
MP-3T	133.00	165.00	155.67	222.00		
MP-3T	141.00	189.00	142.00	181.00		
MP-3T	140.00	187.00	144.00	183.00		
3T Avg	138.00	180.33	147.22	195.33	6.26%	7.68%
MP-4T	188.50	189.00	204.50	219.00		
MP-4T	188.50	190.00	204.75	213.00		
MP-4T	188.00	190.00	204.75	212.00		
4T Avg	188.33	189.67	204.67	214.67	7.98%	11.65%
DK-2T	95.50	96.00	109.50	110.00		
DK-2T	97.00	98.00	107.00	108.00		
DK-2T	96.50	98.00	107.00	108.00		
2T Avg	96.33	97.33	107.83	108.67	10.66%	10.43%
DK-3T	144.33	146.00	161.33	163.00		
DK-3T	144.33	146.00	159.33	160.00		
DK-3T	144.33	145.00	163.00	164.00		
3T Avg	144.33	145.67	161.22	162.33	10.48%	10.27%
DK-4T	192.25	194.00	214.00	215.00		
DK-4T	194.25	196.00	215.25	218.00		
DK-4T	193.50	195.00	213.00	215.00		
4T Avg	193.33	195.00	214.08	216.00	9.69%	9.72%

Effect of adding an additional thread			
	UP	MP	DK
2T-3T	32.96%	31.40%	33.25%
3T-4T	25.26%	26.73%	25.35%

Benefit of DK (TTC)			
	2T	3T	4T
UP	-0.34%	-0.69%	-0.86%
MP	-2.46%	19.22%	-2.81%
UP-NET	4.68%	5.07%	5.40%
MP-NET	0.00%	16.89%	-0.62%

Figure A.4: Vortex Data

Test	No-NET		With-NET		Δ Runtime	Δ TTC
	Avg Runtime	TTC	Avg Runtime	TTC		
UP-2T	188.00	189.00	219.00	220.00		
UP-2T	188.00	189.00	218.50	219.00		
UP-2T	188.00	189.00	218.50	220.00		
2T Avg	188.00	189.00	218.67	219.67	14.02%	13.96%
UP-3T	282.67	284.00	327.00	327.00		
UP-3T	280.67	283.00	327.67	328.00		
UP-3T	281.00	283.00	328.67	330.00		
3T Avg	281.45	283.33	327.78	328.33	14.14%	13.71%
UP-4T	375.25	377.00	439.00	443.00		
UP-4T	375.50	377.00	438.25	442.00		
UP-4T	375.25	378.00	436.75	439.00		
4T Avg	375.33	377.33	438.00	441.33	14.31%	14.50%
MP-2T	175.00	176.00	187.50	201.00		
MP-2T	172.00	173.00	187.50	201.00		
MP-2T	173.00	173.00	187.50	201.00		
2T Avg	173.33	174.00	187.50	201.00	7.56%	13.43%
MP-3T	238.00	295.00	268.00	320.00		
MP-3T	238.67	295.00	277.67	349.00		
MP-3T	255.67	345.00	266.00	317.00		
3T Avg	244.11	311.67	270.56	328.67	9.77%	5.17%
MP-4T	344.75	345.00	375.75	405.00		
MP-4T	348.00	349.00	372.75	402.00		
MP-4T	345.50	349.00	375.75	406.00		
4T Avg	346.08	347.67	374.75	404.33	7.65%	14.01%
DK-2T	188.00	189.00	209.00	210.00		
DK-2T	188.00	190.00	209.00	210.00		
DK-2T	188.00	189.00	209.00	210.00		
2T Avg	188.00	189.33	209.00	210.00	10.05%	9.84%
DK-3T	283.33	285.00	315.67	317.00		
DK-3T	281.67	285.00	314.00	314.00		
DK-3T	281.67	286.00	313.67	314.00		
3T Avg	282.22	285.33	314.45	315.00	10.25%	9.42%
DK-4T	376.75	378.00	419.00	419.00		
DK-4T	377.25	380.00	418.75	419.00		
DK-4T	379.25	382.00	418.25	419.00		
4T Avg	377.75	380.00	418.67	419.00	9.77%	9.31%

Effect of adding an additional thread			
	UP	MP	DK
2T-3T	33.20%	28.99%	33.39%
3T-4T	25.01%	29.46%	25.29%

Benefit of DK (TTC)			
	2T	3T	4T
UP	-0.18%	-0.71%	-0.71%
MP	-8.81%	8.45%	-9.30%
UP-NET	4.40%	4.06%	5.06%
MP-NET	-4.48%	4.16%	-3.63%

Figure A.5: Earthquake Data

Trial	Test	No-NET		With-NET		Δ Runtime	Δ TTC
		Avg Runtime	TTC	Avg Runtime	TTC		
1	UP-2T	328.00	328.00	381.00	382.00		
2	UP-2T	327.00	327.00	371.00	371.00		
3	UP-2T	328.00	328.00	381.00	381.00		
	2T Avg	327.67	327.67	377.67	378.00	13.24%	13.32%
1	UP-3T	492.00	493.00	581.00	581.00		
2	UP-3T	491.00	491.00	583.00	584.00		
3	UP-3T	491.00	491.00	564.00	565.00		
	3T Avg	491.33	491.67	576.00	576.67	14.70%	14.74%
1	UP-4T	660.00	661.00	779.00	779.00		
2	UP-4T	658.00	658.00	771.00	772.00		
3	UP-4T	657.00	658.00	780.00	781.00		
	4T Avg	658.33	659.00	776.67	777.33	15.24%	15.22%
1	MP-2T	436.00	437.00	275.00	288.00		
2	MP-2T	437.00	438.00	277.50	291.00		
3	MP-2T	439.50	441.00	281.00	281.00		
	2T Avg	437.50	438.67	277.83	286.67	-57.47%	-53.02%
1	MP-3T	376.67	422.00	429.33	578.00		
2	MP-3T	374.00	422.00	398.33	446.00		
3	MP-3T	373.67	435.00	417.00	454.00		
	3T Avg	374.78	426.33	414.89	492.67	9.67%	13.46%
1	MP-4T	519.75	523.00	550.75	573.00		
2	MP-4T	519.00	520.00	545.00	568.00		
3	MP-4T	518.00	518.00	549.25	583.00		
	4T Avg	518.92	520.33	548.33	574.67	5.36%	9.45%
1	DK-2T	338.00	346.00	369.00	369.00		
2	DK-2T	335.00	342.00	372.00	372.00		
3	DK-2T	334.00	334.00	357.50	362.00		
	2T Avg	335.67	340.67	366.17	367.67	8.33%	7.34%
1	DK-3T	527.00	528.00	549.00	549.00		
2	DK-3T	507.67	515.00	534.67	547.00		
3	DK-3T	511.00	512.00	553.00	553.00		
	3T Avg	515.22	518.33	545.56	549.67	5.56%	5.70%
1	DK-4T	703.00	703.00	721.25	773.00		
2	DK-4T	672.00	682.00	729.00	729.00		
3	DK-4T	721.00	722.00	720.75	729.00		
	4T Avg	698.67	702.33	723.67	743.67	3.45%	5.56%

	UP	MP	DK
2T-3T	33.31%	-16.74%	34.85%
3T-4T	25.37%	27.78%	26.26%

Benefit of DK			
	2T	3T	4T
UP	-3.97%	-5.42%	-6.58%
MP	22.34%	-21.58%	-34.98%
UP-NET	2.73%	4.68%	4.33%
MP-NET	-28.26%	-11.57%	-29.41%

Figure A.6: Mesa Data

Bibliography

- [1] Cfp2000 benchmark descriptions, 2000.
- [2] Cint2000 benchmark descriptions, 2000.
- [3] 12th International Conference on Parallel Architectures and Compilation Techniques. *The Impact of Resource Partitioning on SMT Processors*, 2003.
- [4] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiawicz, B-H. Lim, K. Mackenzie, and D. Yeung. The MIT alewife machine: Architecture and performance. In *Proc. of the 22nd Annual Int'l Symp. on Computer Architecture (ISCA '95)*, pages 2–13, 1995.
- [5] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst.*, 6(4):393–431, 1988.
- [6] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, 1990.
- [7] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, 1st edition, 2001.
- [8] Control Data Corporation. *6400/6500/6600 Computer Systems Reference Manual*, h edition, 1969.
- [9] David L. Hill Glenn Hinton David A. Koufaty J. Alan Miller Michael Upton Deborah

- T. Marr, Frank Binns. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), February 2002.
- [10] Vincent W. Freeh. Mp scheduler. Private conversation.
- [11] Nicolas Gloy, Cliff Young, J. Bradley Chen, and Michael D. Smith. An analysis of dynamic branch prediction schemes on system workloads. In *Proc. 23rd Annual Intl. Symp. on Computer Architecture*, pages 12–21, 1996.
- [12] ISSCC. *Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading*, volume 1, February 2002.
- [13] Jack L. Lo, Luiz Andre Barroso, Susan J. Eggers, Kouros Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *ISCA*, pages 39–50, 1998.
- [14] Jack L. Lo, Susan J. Eggers, Henry M. Levy, Sujay S. Parekh, and Dean M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *International Symposium on Microarchitecture*, pages 114–124, 1997.
- [15] Robert Love. *Linux Kernel Development*. Sam’s Publishing, 2004.
- [16] Luke K. McDowell, Susan J. Eggers, and Steven D. Gribble. Improving server software support for simultaneous multithreaded processors.
- [17] Dean M. Tullsen Nathan Tuck. Initial observations of the simultaneous multithreading pentium 4 processor. In *Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003.
- [18] Joshua Redstone, Susan J. Eggers, and Henry M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Architectural Support for Programming Languages and Operating Systems*, pages 245–256, 2000.
- [19] A. Snavely. Explorations in symbiosis on two multithreaded architectures, 1999.
- [20] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor, 2002.

- [21] Allan Snaveley and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.
- [22] The BlueGene/L Team. An overview of the bluegene/l supercomputer. Technical report, IBM and Lawrence Livermore National Laboratory, 2002.
- [23] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages ??–??, 1995.
- [24] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ISCA*, pages 191–202, 1996.