UNLV Theses, Dissertations, Professional Papers, and Capstones

8-1-2012

CPU Scheduling for Power/Energy Management on Heterogeneous Multicore Processors

Rajesh Patel
University of Nevada, Las Vegas, patel2@unlv.nevada.edu

CPU SCHEDULING FOR POWER/ENERGY MANAGEMENT ON HETEROGENEOUS MULTICORE PROCESSORS

By

Rajesh Patel

Bachelor of Science in Biological Sciences
University of Southern California
1991

Bachelor of Science in Nuclear Medicine
University of Nevada, Las Vegas
1999

Master of Science in Computer Science
University of Nevada, Las Vegas
2005

A dissertation submitted in partial fulfillment
of the requirements for the

Doctor of Philosophy in Computer Science

School of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
August 2012

UNLV

UNIVERSITY OF NEVADA LAS VEGAS

THE GRADUATE COLLEGE

We recommend the dissertation prepared under our supervision by

Rajesh Patel

entitled

CPU Scheduling for Power/Energy Management on Heterogeneous Multicore Processors

be accepted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science
School of Computer Science

Ajoy K. Datta, Committee Chair

Lawrence Larmore Committee Member

Yoohwan Kim, Committee Member

Venkatesan Muthukumar, Committee Member

Wolfgang Bein, Graduate College Representative

Tom Piechota, Ph.D., Interim Vice President for Research &
Dean of the Graduate College

August 2012

ABSTRACT

CPU Scheduling for Power/Energy Management on Heterogeneous Multicore Processors

by

Rajesh Patel

Dr. Ajoy K. Datta, Examination Committee Chair
School of Computer Science
University of Nevada, Las Vegas

Power and energy have become increasingly important concerns in the design and implementation of today's multicore/manycore chips. Many methods have been proposed to reduce a microprocessor's power usage and associated heat dissipation, including scaling a core's operating frequency. However, these techniques do not consider the dynamic performance characteristics of an executing process at runtime, the execution characteristics of the entire task to which this process belongs, the process's priority, the process's cache miss/cache reference ratio, the number of context switches and CPU migrations generated by the process, nor the system load. Also, many of the techniques that employ dynamic frequency scaling can lower a core's frequency during the execution of a non-CPU intensive task, thus lowering performance. In addition, many of these methods require specialized hardware and have not been tested upon real hardware that is widely available, including the recent AMD or Intel multicore chips.

One problem dealing with power/energy management for heterogeneous multicore processors is: Given a set of processes, each having identical default priorities, in a given task to be executed by a heterogeneous multicore/manycore processor system, schedule each process in this task to execute upon the CPU(s) in this system such that the global power budget is minimized, yet the performance gain of all processes is maximized, and the performance loss of all processes is minimized. Doing so, in a scenario where each process has a different (not necessarily unique) static or dynamic (but not necessarily the default) priority, without adversely affecting process completion order, as dictated by process priority is yet another problem. Finally, utilizing the cache miss/cache reference ratio and the number of context switches and CPU migrations as scheduling criteria are two other problems. This dissertation will elaborate upon these four problems, and will describe our four approaches to solving these problems.

ACKNOWLEDGMENTS

I would like to offer a special thanks to my dissertation advisor, Dr. Ajoy K. Datta, for chairing my committee and advising me throughout my dissertation work. His patience, support, enthusiasm, and most importantly, his confidence in my abilities, have helped me greatly throughout my graduate study. I am also very grateful to Dr. Wolfgang Bein, Dr. Lawrence Larmore, Dr. Yoohwan Kim, and Dr. Venkatesan Muthukumar for their participation in my committee.

This dissertation is dedicated to my parents and family. Their support, love, and faith in my abilities were very important for completing this dissertation research.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS v

LIST OF TABLES viii

LIST OF FIGURES x

CHAPTER 1 INTRODUCTION 1

Contributions 3

Outline of the Dissertation 4

CHAPTER 2 MULTICORE PROCESSORS 6

Overview 6

Development 7

Overview of Architecture 10

Homogeneous versus Heterogeneous Multicore Processors 11

CHAPTER 3 MULTICORE PROCESSOR IMPLEMENTATION CHALLENGES 13

Cache Coherence 13

Power and Temperature Management 14

Multicore Performance 16

CHAPTER 4 POWER/ENERGY CPU SCHEDULING PROBLEM FOR HETEROGENEOUS MULTICORE PROCESSOR SYSTEMS . . 18

Motivation 18

Related Work 20

Problem Specification 23

CHAPTER 5 ALGORITHM 1: Algorithm *CPU Scheduler* 25

Variables and Constants Used in Algorithm *CPU Scheduler* 25

Description of Algorithm *CPU Scheduler* 27

Algorithm 1 Feedback-driven CPU Scheduling Algorithm (*CPU Scheduler*) . . . 31

CHAPTER 6 ALGORITHM 2: Algorithm *Priority CPU Scheduler* 33

Variables and Constants Used in Algorithm *Priority CPU Scheduler* 33

Description of Algorithm *Priority CPU Scheduler* 35

Algorithm 2 Feedback-driven Priority-based CPU Scheduling Algorithm (*Priority CPU Scheduler*) 40

CHAPTER 7 ALGORITHM 3: Algorithm *Cache Miss Priority CPU Scheduler* 42

Variables and Constants Used in Algorithm *Cache Miss Priority CPU Scheduler* 42
Description of Algorithm *Cache Miss Priority CPU Scheduler* 44
Algorithm 3 Cache Miss Feedback-driven Priority-based CPU Scheduling Algorithm (*Cache Miss Priority CPU Scheduler*) 49

CHAPTER 8 ALGORITHM 4:
Algorithm *Context Switch Priority CPU Scheduler* 51
Variables and Constants Used in Algorithm *Context Switch Priority CPU Scheduler* 51
Description of Algorithm *Context Switch Priority CPU Scheduler* 53
Algorithm 4 Context Switch Feedback-driven Priority-based CPU Scheduling Algorithm (*Context Switch Priority CPU Scheduler*) 59

CHAPTER 9 EVALUATION AND RESULTS 61
Methodology 61
Discussion of Results 63
Evaluation of Algorithm *CPU Scheduler* 64
Evaluation of Algorithm *Priority CPU Scheduler* 72
Evaluation of Algorithm *Cache Miss Priority CPU Scheduler* 94
Evaluation of Algorithm *Context Switch Priority CPU Scheduler* 105

CHAPTER 10 CONCLUSION AND FUTURE RESEARCH 115

BIBLIOGRAPHY 118

VITA 121

LIST OF TABLES

Table 1 Benchmarks and Tasks Evaluated 64

Table 2 Performance of Benchmarks Evaluated for Algorithm 1 65

Table 3 Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 67

Table 4 Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 69

Table 5 Completion Order of Benchmarks Evaluated (static priorities noncpuintensive benchmarks higher priority) 74

Table 6 Completion Order of Benchmarks Evaluated (static priorities noncpuintensive benchmarks lower priority) 74

Table 7 Completion Order of Benchmarks Evaluated (dynamic priorities) 74

Table 8 Completion Order of Benchmarks Evaluated for Task 4 75

Table 9 Performance of Benchmarks Evaluated (static priorities noncpuintensive benchmarks higher priority) for Algorithm 2 78

Table 10 Performance of Benchmarks Evaluated (static priorities noncpuintensive benchmarks lower priority) for Algorithm 2 78

Table 11 Performance of Benchmarks Evaluated (dynamic priorities) for Algorithm 2 78

Table 12 Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (static priorities noncpuintensive benchmarks higher priority) 84

Table 13 Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (static priorities noncpuintensive benchmarks lower priority) 84

Table 14 Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (dynamic priorities) 84

Table 15 Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (static priorities noncpuintensive benchmarks higher priority) 89

Table 16 Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (static priorities noncpuintensive benchmarks lower priority) 89

Table 17 Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time Watt for Tasks 1, 2, and 3 (dynamic priorities) 89

Table 18 Average Number of Cache Misses, Cache References, and Average Cache Miss/Reference Ratio for Algorithms 3, 2, and Control 97

Table 19 Performance, Performance per Watt, and Execution Time Watt for Task 4 (dynamic priorities) 100

Table 20 Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt by Algorithm 3 for Task 4 (dynamic priorities) 100

Table 21 Percent Reduction of Average Number of Cache Misses, Cache References, and Average Cache Miss/Reference Ratio over Control by Algorithms 3 and 2 and over Algorithm 2 by Algorithm 3 103

Table 22 Average Number of Context Switches and CPU Migrations for Algorithms 4, 2, and Control 108

Table 23 Performance, Performance per Watt, and Execution Time · Watt for Task 4 (dynamic priorities) 110

Table 24 Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt by Algorithm 4 for Task 4 (dynamic priorities) 110

Table 25 Percent Reduction of Average Number of Context Switches and CPU Migrations over Control by Algorithms 4 and 2 and over Algorithm 2 by Algorithm 4 113

LIST OF FIGURES

Figure 1 General Overview of Algorithms 1 - 4 4

Figure 2 Basic Microprocessor Design 8

Figure 3 Shared Memory Model (left) and Distributed Memory Model (right) 9

Figure 4 Block Diagrams of the Core 2 Duo and Athlon 64 X2 multicore processors 10

Figure 5 Block Diagram of AMD Opteron 6100 processor 11

Figure 6 Relationship Between Core Operating Frequency and Power Consumption 14

Figure 7 Hardware Partitioning 62

Figure 8 Performance of Benchmarks Evaluated for Algorithm 1 66

Figure 9 Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 68

Figure 10 Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 70

Figure 11 Completion Order of Benchmarks Evaluated 75

Figure 12 Performance of Benchmarks Evaluated (static priorities noncpuintensive benchmarks higher priority) for Algorithm 2 79

Figure 13 Performance of Benchmarks Evaluated (static priorities noncpuintensive benchmarks lower priority) for Algorithm 2 80

Figure 14 Performance of Benchmarks Evaluated (dynamic priorities) for Algorithm 2 81

Figure 15 Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (static priorities noncpuintensive benchmarks higher priority) 85

Figure 16 Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (static priorities noncpuintensive benchmarks lower priority) 86

Figure 17 Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, and 3 (dynamic priorities) 87

Figure 18 Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (static priorities – noncpuintensive benchmarks higher priority) 90

Figure 19 Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (static priorities – noncpuintensive benchmarks lower priority) 91

Figure 20 Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, and 3 (dynamic priorities) 92
Figure 21 Average Number of Cache Misses, Cache References, and Cache Miss/Reference Ratio for Algorithms 3, 2, and Control for Task 4 (dynamic priorities) 98
Figure 22 Performance, Performance per Watt, Execution Time · Watt, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt by Algorithm 3 for Task 4 (dynamic priorities) 101
Figure 23 Percent Reduction of Average Number of Cache Misses, Cache References, and Average Cache Miss/Reference Ratio over Control by Algorithms 3 and 2 and over Algorithm 2 by Algorithm 3 104
Figure 24 Average Number of Context Switches and CPU Migrations for Algorithms 4, 2, and Control for Task 4 (dynamic priorities) 108
Figure 25 Performance, Performance per Watt, Execution Time · Watt, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt by Algorithm 4 for Task 4 (dynamic priorities) 111
Figure 26 Percent Reduction of Average Number of Context Switches and CPU Migrations over Control by Algorithms 4 and 2 and over Algorithm 2 by Algorithm 4 113

CHAPTER 1

INTRODUCTION

Recent advances in processor production have led to not only an increase in the number of cores on a single chip, but also a concomitant increase in power consumption and heat dissipation associated with these high performance systems. For example, a 2.8 GHz Intel "Northwood" Pentium 4 chip consumes around 80 W, while a "Prescott" Pentium 4 chip of the same speed consumes around 100 W [24]. This increased power consumption creates an associated increase in heat dissipation, which leads to higher costs for air conditioning, thermal packaging, fans, and electricity. Increased heat and temperature of hardware, such as the CPU, can also lead to decreased longevity and greater incidence of failure of these components. Also, the integration and design of a multicore chip is more difficult to manage, than a lower-density, single-chip design due to thermal constraints [23]. Finally, power savings and decreased heat production can be crucial in notebook computers, where battery life is a constraint and cooling of hardware components is difficult, and in server farms, where even a slight decrease in the power consumption of each server can cause a significant savings for the entire system.

Scaling a CPU's frequency can dramatically affect its power consumption. In fact, the power dissipation at the output pins of a core is directly proportional to its frequency and is governed by the equation

$$P = \frac{1}{2}CV^2f$$

where f is the effective bus frequency.

CPU frequency scaling may be used to produce an architecture composed of a heterogeneous processor, those with cores operating at different frequencies. Heterogeneous architectures may achieve a higher performance per watt than comparable homogeneous systems [30, 21] due to the ability of each application to run on a core that best suits its architectural properties. An architecture composed of a heterogeneous processor may contain cores specialized for certain tasks. For example, threads conducting CPU intensive work may execute upon high frequency cores, while threads conducting memory intensive work may execute upon low frequency cores, which consume significantly less power.

Memory intensive processes may suffer from memory bandwidth saturation [8, 14]. During the execution of such processes, if data cannot be moved from main memory to the cores fast enough, the cores may sit idle as they wait for the data to arrive. These processes may obtain limited benefit from increasing processor frequency due to the slow speed of cache and memory access relative to processor speed. Thus when executing a task containing both CPU intensive and memory intensive processes, it may be beneficial to move such memory intensive processes to lower frequency cores, while allowing less memory intensive, more CPU intensive processes to execute upon higher frequency cores.

The Linux 2.6 real time task scheduler minimizes response times for critical real-time tasks, while maximizing CPU utilization, by implementing dynamic task prioritization and task preemption [16]. That is, to prevent tasks from starving other tasks that need to use the CPU, the Linux 2.6 scheduler can dynamically alter a task's priority. A task's priority is dictated by its nice value, with a lower nice value indicating a higher priority. Also the Linux 2.6 scheduler allows preemption, meaning a lower priority task won't execute when a higher priority task is ready to run. By default, all processes have a nice value of 0, thus having the same priority. However, tasks may be assigned different priorities, based upon their nice values. This may affect the completion order of a group of concurrently executing tasks.

Context switching less CPU intensive processes, that would benefit less from executing upon higher frequency cores, to lower frequency cores and allowing CPU intensive processes to execute upon higher frequency cores may minimize performance loss while providing significant power savings. Doing so in a manner that does not affect process completion order as dictated by process priority is also beneficial. This research's goal is to identify and quantify these advantages.

1.1 Contributions

Like some recent work using heterogeneous processors [14, 5, 32, 2, 17, 30, 35, 20, 19, 21, 11, 33, 25, 26, 10, 13, 22], this research uses processors that have the same instruction architecture but have different implementation characteristics, such as operating frequency. In contrast to these works, however, our paper offers the following contributions.

We present four CPU scheduling algorithms, referred to in this paper as Algorithm *CPU Scheduler*, Algorithm *Priority CPU Scheduler*, Algorithm *Cache Miss Priority CPU Scheduler*, and Algorithm *Context Switch Priority CPU Scheduler* that lower the global power budget in a heterogeneous multicore (or manycore) system while creating a minimal performance loss (and in some cases a performance gain) given a task containing a set of processes to be executed on this system. Also, for the three latter algorithms, process completion order, as dictated by process priority, is not affected. These algorithms utilize hardware partitions composed of heterogeneous cpusets. The cpusets contain varying numbers of cores. The cores are identical, with the exception that cores belonging to the same cpuset operate at the same frequency, and cores belonging to different cpusets operate at different frequencies. The algorithms can be executed upon a system using any multicore/manycore chip supporting CPU frequency scaling, which includes most of the widely available chips, including the Intel Nehalem and AMD Opteron chips, and uses no specialized hardware or software.

Algorithms *CPU Scheduler*, *Priority CPU Scheduler*, *Cache Miss Priority CPU Scheduler*, and *Context Switch Priority CPU Scheduler* use “application-driven” feedback, in which the executing application gives feedback (regarding its performance) to the operating system, at runtime, and the operating system, in turn, schedules the system’s CPU hardware resources based upon this feedback. At runtime, an application evaluates its performance and sends this feedback to the operating system. If the application has an increased performance, relative to other executing processes in the task, the operating system context switches the application to a cpuset containing cores operating at a lower frequency, whereas if an application has decreased performance, relative to other executing

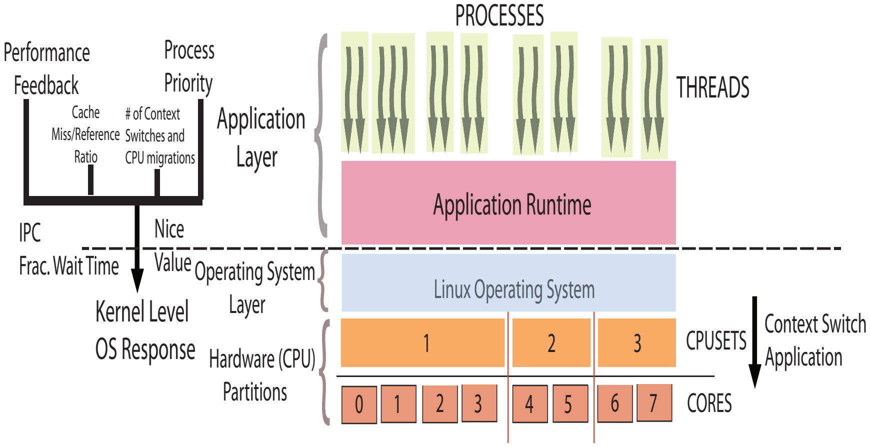


Figure 1: General Overview of Algorithms 1 - 4

processes in the task, the operating system context switches the application to a cpuset containing cores operating at a higher frequency, in hopes of speeding up the execution of the application and thus lowering the average completion time of all processes in the task. In addition, the latter three algorithms use process priority (Linux nice values), both static and dynamic, as a scheduling criterion and do not affect process completion order in a scenario where each process is assigned a distinct nice value. Also, algorithms *Cache Miss Priority CPU Scheduler* and *Context Switch Priority CPU Scheduler* use the cache miss/cache reference ratio and the number of context switches and CPU migrations, respectively, as scheduling criteria with the goal of improving performance. A general overview of our algorithms is depicted in Figure 1.

Power reduction is achieved by utilizing the heterogeneous cpusets, many of which contain cores operating at a lower frequency than would be achieved if the same cores were executing the same processes using the "on demand" CPU frequency scaling governor, which is the default governor in a Linux based system. This results in significant power savings.

1.2 Outline of the Dissertation

We start with a discussion of the design of multicore processors in Chapter 2. This includes an overview of the concept of multicore processors. We then discuss the events leading to the development of multicore processors. We also present an overview of the architecture of multicore processors. Finally, we explain the concepts of homogeneous versus heterogeneous multicore processors. In Chapter 3, we discuss challenges associated with the implementation of multicore processors. We include a discussion of cache coherence, power and temperature management, and performance issues arising from the implementation of multicore processors. In Chapter 4 we state the motivation of this research, describe some results in related areas, and introduce our four CPU scheduling problems. The main contribution of this dissertation is presented in Chapters 5, 6, 7, and 8 where we present our solutions to the four problems, including descriptions of the variables and constants used in the algorithms, descriptions of the algorithms themselves, and formal presentations of our algorithms. The first algorithm, Algorithm *CPU Scheduler*, is presented in Chapter 5, the second, Algorithm *Priority CPU Scheduler* , is presented in Chapter 6, the third, Algorithm *Cache Miss Priority CPU Scheduler* , is presented in Chapter 7, and the fourth, Algorithm *Context Switch Priority CPU Scheduler*, is presented in Chapter 8. A discussion of the complexity of the algorithms, evaluation results, and other properties are included in Chapter 9. Finally, we conclude and present some ideas for future research in Chapter 10.

CHAPTER 2

MULTICORE PROCESSORS

2.1 Overview

A multicore processor is an integrated circuit or single computing component upon which two or more independent processors (cores) have been attached for enhanced performance, reduced power consumption, and parallel processing. A multicore processor is comparable to having two or more separate processors installed on the same computer. However, because these processors are plugged into the same socket, the connection between them is faster.

The cores are units that read and execute CPU instructions such as add, move data, and branch. The multiple cores can execute multiple instructions simultaneously, thus increasing the overall speed of programs that can take advantage of parallel computing. These cores are typically integrated onto a single integrated circuit die, known as a chip multiprocessor (CMP), or onto multiple dies in a single chip package [23]. AMD, ARM, Broadcom, Intel, and VIA are among the companies that have produced or are currently working on multicore products.

A dual core processor has two cores (e.g. AMD Phenom II X2), a quad core processor has four cores (Intel i3, i5, i7), a hexa core processor contains six cores (AMD Phenom II X6), and an octa core processor contains eight cores (AMD Opteron 6134). Chips with tens or even hundreds of cores are the growing trend in processor development. Such chips are termed “manycore” chips. In a manycore processor, the number of cores is large enough such that traditional multi-processor techniques are not efficient, primarily due to issues with congestion in supplying instructions and data to all of the cores. In this case, network on chip technology is advantageous.

The cores in a multicore device can be coupled tightly or loosely by designers. One example is that cores may or may not share caches, and may utilize message passing or shared memory for inter-core communication. Common network topologies interconnecting cores include bus, ring, two-dimensional mesh, and crossbar. Also, as in single-processor systems, cores in a multicore system may utilize architectures such as superscalar, VLIW, vector processing, SIMD, or multithreading.

2.2 Development

Because single core processors are quickly reaching the physical limits of feasible complexity and speed, multicore processing is becoming a growing industry trend [29]. While various methods have been used to improve CPU performance, including instruction level parallelism (ILP) methods such as superscalar pipelining, many of these methods are inefficient for applications that contain code that is hard to predict. These applications can benefit more from thread level parallelism (TLP) methods. Multiple independent CPUs are commonly used to increase a system's overall TLP. Thus one contributing factor that led to the development of multicore processors is the demand for increased TLP.

The speedup of a program using a multicore processor in parallel computing is quantified using Amdahl's law. Amdahl's law states that the overall speedup of a program is

$$S = \frac{1}{(1-P)+\frac{P}{n}}$$

where P is the portion of the program that can be executed in parallel and n is the number of concurrent processors.

It was possible for decades to improve the performance of a CPU by shrinking the area of the integrated circuit. Also, for the same circuit area, more transistors could be utilized in the design, which increased functionality. Clock rates also increased in the late 20^{th} century, from several megahertz in the 1980's to several gigahertz in the early 2000's. However, as the rate of speeding up processor frequency had slowed and reached a plateau, increased use of parallel computing in the form of multicore processors has been pursued to improve overall processing performance. Adding additional processor cores to the same chip should both

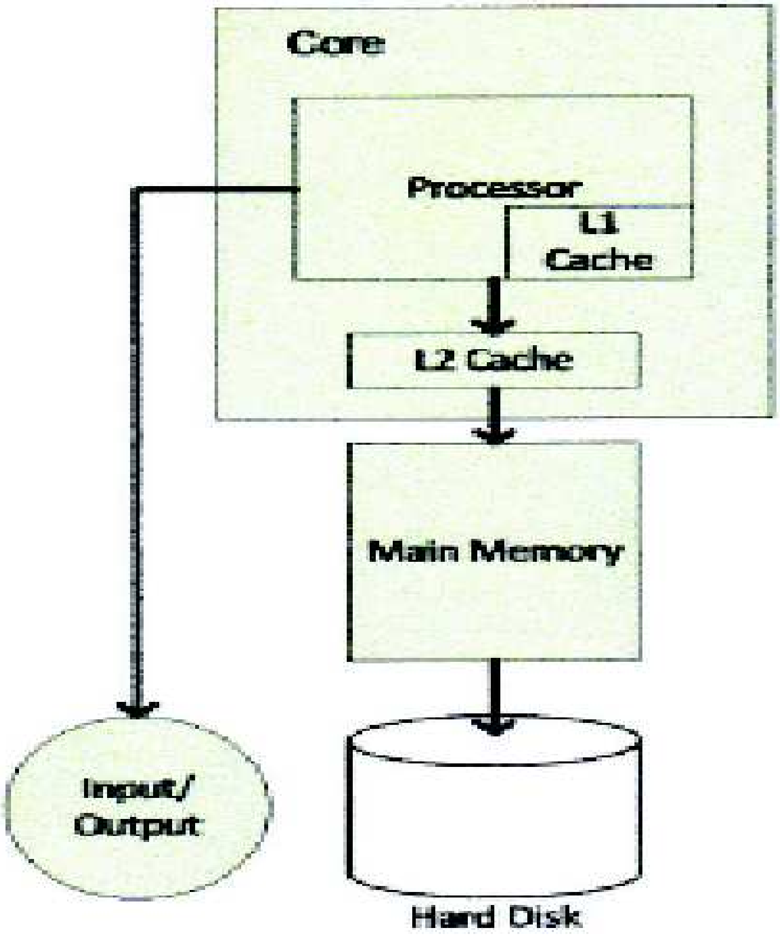


Figure 2: Basic Microprocessor Design

improve performance and dissipate less heat, “without the need to run at ruinous clock rates [18]”. Thus both increased available space (due to improved manufacturing processes) and the demand for increased thread level parallelism have led to the development of multicore CPUs.

Today, multicore processors are used in a wide variety of applications, including general purpose, embedded, network, digital signal processing (DSP), and graphics.

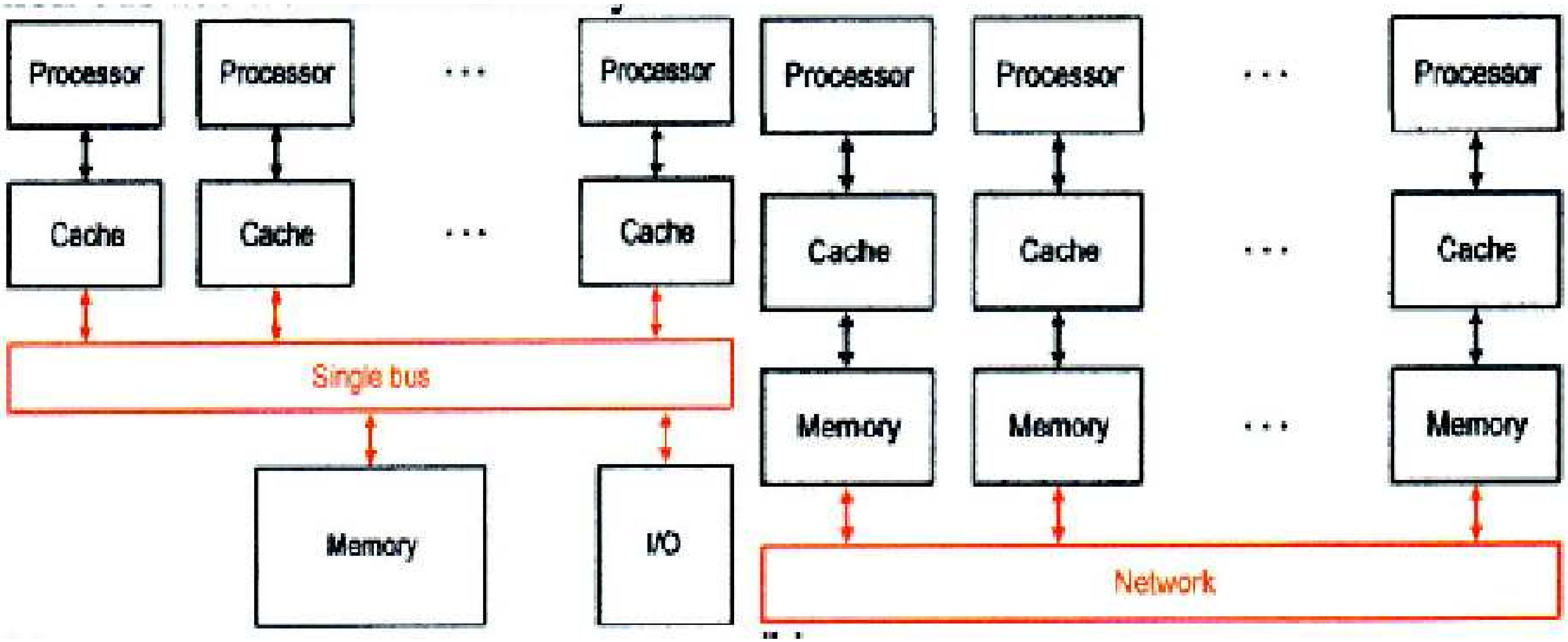
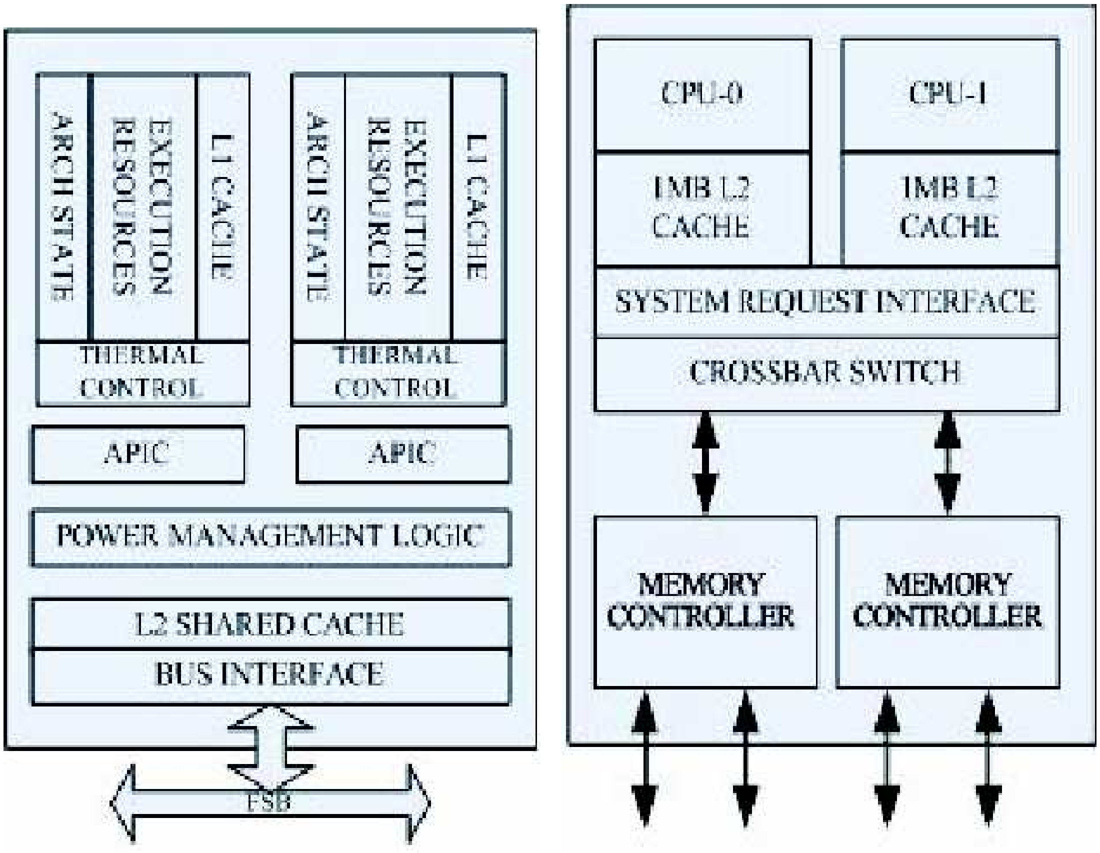


Figure 3: Shared Memory Model (left) and Distributed Memory Model (right)



(a) Block Diagram of Intel Core 2 Duo

(b) Block Diagram of AMD Athlon 64 X2

Figure 4: Block Diagrams of the Core 2 Duo and Athlon 64 X2 multicore processors

2.3 Overview of Architecture

A basic overview of multicore architecture is as follows. The basic design of a microprocessor is depicted in Figure 2. Level 1 (L1) cache is very fast memory that is used to store data frequently used by a processor. It is also physically closest to a processor. Level 2 (L2) cache is just off-chip, is larger than L1 cache, and is slower than L1 cache, but is still much faster than main memory. It is used for the same purpose as L1 cache. Most systems have approximately 32 Kb of L1 cache and 2 Mb of L2 cache.

Communication between cores, and to main memory, is usually accomplished by using either a single communication bus or and interconnection network. A bus is used with a shared memory model and an interconnection network is used with a distributed memory model. A bus has limited scalability and leads to diminished performance after the number

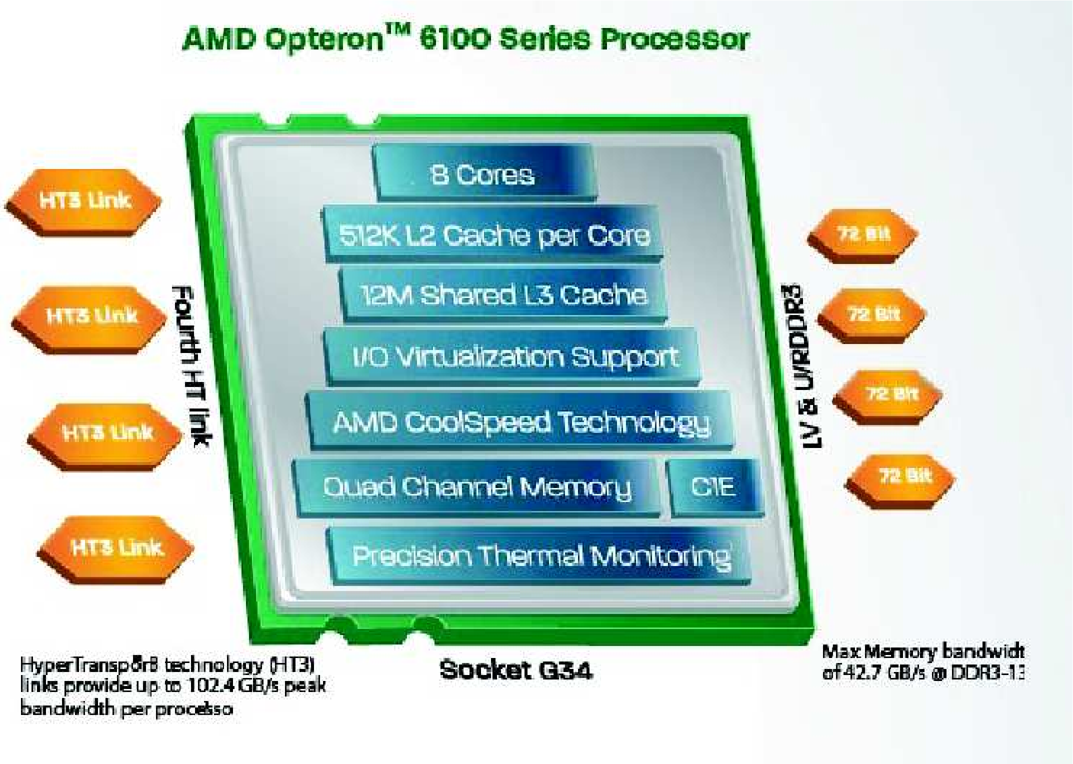

AMD Opteron™ 6100 Series Processor
8 Cores
512K L2 Cache per Core
12M Shared L3 Cache
I/O Virtualization Support
AMD CoolSpeed Technology
Quad Channel Memory
C1E
Precision Thermal Monitoring
HT3 Link
HT3 Link
HT3 Link
HT3 Link
Fourth HT link
LV & U/RDDR3
72 Bit
72 Bit
72 Bit
72 Bit
HyperTranspōr8 technology (HT3) links provide up to 102.4 GB/s peak bandwidth per processo
Socket G34
Max Memory bandwidt of 42.7 GB/s @ DDR3-13

Figure 5: Block Diagram of AMD Opteron 6100 processor

of cores is approximately thirty-two cores. Figure 3 depicts both the shared memory and distributed memory models.

Two examples of different multicore architectures are the Intel Core 2 Duo chip and the Advanced Micro Device's Athlon 64 X2 chip. Both architectures differ greatly. Figure 4 shows the architecture of both chips [27].

The Core 2 Duo uses a shared memory model with private L1 caches and a shared L2 cache, whereas the Athlon utilizes a distributed memory model with discrete L2 caches for each core. These L2 caches share a system request interface, and in doing so, eliminate the need for a bus. Figure 5 shows a block diagram of the AMD Opteron 6100 processor, which also uses per core L2 caches and is the processor used in this research. The Core 2 Duo, instead, uses a bus interface. It also has explicit thermal and power control units on chip.

2.4 Homogeneous versus Heterogeneous Multicore Processors

Homogeneous multicore systems are systems with identical cores and use one core design that is repeated consistently. A heterogeneous multicore processor is one which uses a combination of different cores, each of which can be optimized for a different role. One area of heterogeneity can be a core's operating frequency.

A single-ISA heterogeneous multicore processor, or simply heterogeneous multicore processor, can consist of cores exposing the same instruction set architecture, but delivering different performance. The cores can differ in clock frequency and power consumption. Asymmetry (or heterogeneity) can be built in by design [20, 21], or may exist due to explicit clock frequency scaling [31]. The latter is the basis of the heterogeneity in our heterogeneous multicore multiprocessor system.

CHAPTER 3

MULTICORE PROCESSOR IMPLEMENTATION CHALLENGES

There are numerous challenges that arise with the implementation of a multicore processor system. Three of these, which will be discussed in this chapter, are cache coherence, power and heat dissipation, and performance.

3.1 Cache Coherence

Cache coherence can be an issue with multicore processors because of distributed L1 and L2 caches. The copy of the data in a core's cache may not always be the most up-to-date version since each core has its own cache. An example of this may be a dual-core processor where each core brought a block of memory into its private cache. If one core writes a value to a specific location, when the second core tries to read that value from its cache, it won't have the updated copy unless its cache entry is invalidated and a cache miss occurs. This cache miss causes the second core's cache entry to be updated. Without this type of cache coherence policy, garbage data would be read and invalid results would be produced.

Generally, there are two schemes for cache coherence, a directory-based protocol and a snooping protocol. The directory-based protocol can be used on an arbitrary network, and therefore is scalable to many processors or cores. In this scheme, a directory is used that holds information about which memory locations are being shared in multiple caches and which are exclusively used by one core's cache. This directory knows when a block needs to be invalidated or updated. In contrast, the snooping protocol only works with a bus-based system, and uses a number of states to decide whether it needs to update cache entries and if it has control over writing to a block. Also, the snooping protocol is not scalable [12].

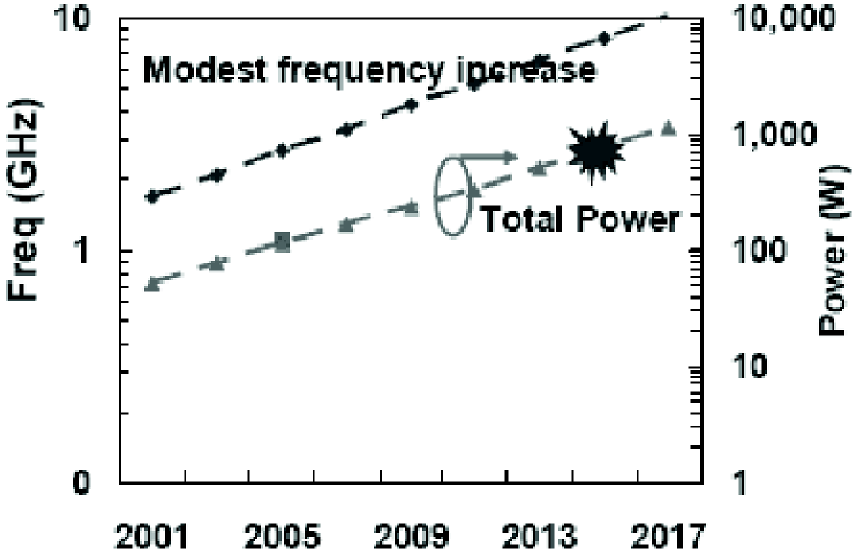


Figure 6: Relationship Between Core Operating Frequency and Power Consumption

Two different schemes of cache coherence are exemplified by Intel's Core 2 Duo and AMD's Athlon 64 X2 multicore chips. Intel's Core 2 Duo speeds up cache coherence by querying the second core's L1 cache and the shared L2 cache simultaneously. An extra benefit of a shared L2 cache is that a coherence protocol doesn't need to be set for this level. AMD's Athlon 64 X2 monitors cache coherence in both L1 and L2 caches. This method has more overhead than Intel's model, despite its attempt to speed this process up using the HyperTransport connection.

This overhead associated with cache coherence can degrade the performance of a multicore processor system. Also, if a process is context switched too often, it will also have a tendency to generate many cache misses since it changes the CPU upon which it executes frequently and may not find valid data in its new core's cache. As a result, a performance loss will also occur, due to the need for cache coherence and the need to fetch valid data from another core's cache. Thus a different strategy may need to be implemented if a process generates many cache misses during an algorithm's implementation.

3.2 Power and Temperature Management

The design and implementation of today's multicore/manycore chips must consider power and heat dissipation as important issues. In theory, if two cores are placed on a single chip, without any further modification, the chip would consume twice as much power [27]. This increased power consumption creates an associated increase in heat dissipation. As mentioned earlier, thermal constraints also make the integration and design of a multicore chip difficult. This necessitates a method of power reduction and temperature management for a multicore chip upon which numerous power consuming cores are found.

One method to account for the large amount of heat produced by a multicore/manycore chip is to run the multiple cores at a lower frequency to reduce power consumption. Figure 6 shows the linear relationship between a core's operating frequency and its power consumption. One technique to lower a core's frequency is dynamic voltage and frequency scaling, or DVFS. By using this technique, the operating voltage and frequency of a processor can be decreased. DVFS is used to decrease the voltage and frequency of a processor in order to conserve power, particularly in laptops or other mobile devices [9]. With this decrease in power consumption comes a concomitant decrease in heat production. Due to the quadratic relation between the energy consumption and operating voltage and linear relation between the energy consumption and operating frequency of a microprocessor, the DVFS technique has been proven to be a highly effective method of achieving low power consumption for a CPU while maintaining performance requirements [6].

CPU underclocking is another technique in which a CPU can be set to run at a lower clock rate than it was specified to operate at to reduce a computer's power consumption and heat emission [34]. This technique can provide increased system stability in high heat environments. An example is a Pentium 4 processor clocked at 2.4 GHz that can be "underclocked" to 1.8 GHz and can then be run with reduced fan speeds. In fact, in laptops the processor is usually underclocked automatically when the computer is operating on batteries. Most recent notebook and desktop computers utilizing power-saving schemes, like AMD's PowerNow and Cooln'Quiet, underclock themselves automatically under a light processing load when the system BIOS and operating system support it [3].

Yet another power management technique for multicore chips is to incorporate a power management unit that has the authority to shut down unused cores or limit the amount of power to these cores. Also, some processors, like the Intel Core 2 Duo Notebook processor, can turn off parts of the CPU, which are not used, to save energy [15].

Two of the techniques mentioned in this section, DVFS and underclocking, scale a CPU's frequency. The algorithms presented in this paper also rely upon CPU frequency scaling to achieve power savings.

3.3 Multicore Performance

Multicore chips can contribute to increased performance of a processor, but this benefit can only exist if parallelism is exploited. Multicore chips improve an operating system's ability to multitask applications. In other words, multiple tasks can run in parallel upon the cores of a multicore chip, thereby improving overall performance.

A related topic is multithreading, or other parallel processing technique to get the optimum performance from a multicore processor. "With the possible exception of Java, there are no widely used commercial development languages with [multithreaded] extensions." [7] Programmers often have to completely rework and rebuild applications to make them multithreaded. They have to write applications with subroutines able to be run upon different cores. This means that data dependencies have to be resolved or accounted for such as race conditions, communication latencies, or cache coherence. Also, multithreaded applications must be balanced, meaning one core should not be used much more than another, in order to take full advantage of multithreading in a multicore system.

However, through the use of parallel programming and multithreading, these challenges in multicore performance can often be overcome. Also, in an architecture composed of a heterogeneous multicore processor, each application and thread can execute upon a core that best suites its architectural properties, thus overcoming another multicore performance challenge. An architecture composed of a heterogeneous multicore processor may contain cores specialized for certain tasks. For example, threads conducting CPU intensive work may execute upon high frequency cores, while threads that are not CPU intensive may execute upon low frequency cores. In this manner, threads that can benefit the most from executing upon higher frequency cores, may do so, while non-CPU intensive threads can execute upon lower frequency cores. Examples of non-CPU intensive threads or applications are those executing many system calls or interrupts, those that are I/O bound, and memory intensive processes.

Finally, the last challenge with multicore performance can be overcoming the performance loss associated with context switching overhead. In a multicore/manycore processor, an application can often be context switched upon the cores of a multicore processor. These context switches and CPU migrations are associated with a context switching overhead. This overhead can occur from the actual migration of a process's state from one core to another, or the delay associated with cache (memory) access and cache coherence. If a process is context switched too often, it will have a tendency to generate many cache misses, since it is changing the CPU upon which it executes frequently, and each CPU or core often has its own L1 or L2 cache.

To overcome this challenge, Algorithms 3 and 4 presented in this paper, consider the cache miss/cache reference ratio and the number of context switches and CPU migrations as scheduling criteria. Thus, if a certain threshold for these criteria is reached, indicating that an executing process is undergoing a performance loss due to an increase of cache misses, context switches, or CPU migrations, then a different strategy is implemented. These strategies will be described in the next few chapters.

CHAPTER 4

POWER/ENERGY CPU SCHEDULING PROBLEM FOR HETEROGENEOUS MULTICORE PROCESSOR SYSTEMS

After extensively researching power concerns in the implementation of heterogeneous multiprocessor systems, we designed four algorithms concerned with lowering the global power budget in a heterogeneous multicore processor system while creating a minimal performance loss, and, in some cases, a performance gain. We state the motivation of this research in the next section. We state how other problems mentioned in earlier chapters are related to the problem solved in this chapter. Finally, we give both an informal explanation and formal statement of the problems to be solved.

The main results of this dissertation research are reported in the next four chapters. In Chapters 5, 6, 7, and 8 four CPU scheduling algorithms (Algorithm 1: *CPU Scheduler*, Algorithm 2: *Priority CPU Scheduler*, Algorithm 3: *Cache Miss Priority CPU Scheduler* , and Algorithm 4: *Context Switch Priority CPU Scheduler*), each with a different approach to lowering the global power budget of a heterogeneous multicore processor system, are presented. In each of these four chapters, we include a description of the variables and constants used in our algorithm, a detailed informal description of the algorithm itself, and a formal presentation of our algorithm. Evaluation results and other properties of all four algorithms are given in Chapter 9.

4.1 Motivation

A novel approach to allocating CPU resources with the goal of lowering the global power budget and creating a minimal performance loss (or a performance gain) in a heterogeneous multicore processor system is to use “application-driven” feedback, in which an executing application gives feedback (regarding its performance) to the operating system at runtime, and the operating system, in turn, dynamically schedules the system’s CPU hardware resources based upon this feedback. There are four approaches presented in this dissertation that are concerned with meeting this goal.

In the first approach, each process in the system has the same default Linux nice value of 0, and thus the same default priority. This approach does not use process priority as a variable. In the second approach, each process in the system has different (not necessarily unique) priorities. These priorities will be implemented using Linux nice values, and may or may not be the default nice value of 0. In addition, they may be static or dynamic. Also, if a process’s priority is static, different trial types will be used in which a process will be assigned both high and low priorities. This approach uses a process’s nice value as a variable. In the third approach, in addition to each process having dynamic priorities, the cache miss/reference ratio of a process at runtime is used as a scheduling factor. Finally, in the fourth approach, each process has a dynamic priority and the number of context switches and CPU migrations generated by the process at runtime is used as a scheduling factor.

Our approaches will utilize hardware partitions composed of heterogeneous cpusets. The cpusets will contain varying numbers of cores. The cores will be identical, with the exception that cores belonging to the same cpuset will operate at the same frequency, and cores belonging to different cpusets will operate at different frequencies.

Power savings will be achieved by utilizing the heterogeneous cpusets, many of which will contain cores operating at a lower frequency than would be achieved if the same cores were executing the same processes using the “on demand” CPU frequency scaling governor, which is the default governor in a Linux based system.

To compensate for the lack of parallelism available in cpusets containing fewer cores, the cpusets will have different numbers of CPU’s, with the number of CPU’s in a cpuset being inversely proportional to the frequency of its cores.

Application performance will be improved due to two main factors. Firstly, if an application has an increased performance, relative to the other executing processes in a task, the operating system will context switch the application to a cpuset containing cores operating at a lower frequency, whereas if an application has decreased performance, relative to other executing processes in the task, the operating system will context switch the application to a cpuset containing cores operating at a higher frequency, in hopes of speeding up the execution of the application and thus lowering the average completion time of all processes in the task. Secondly, by context switching processes between cpusets, the algorithm will move memory intensive processes that may suffer form memory bandwidth saturation to lower frequency cores, while allowing less memory intensive, more CPU intensive processes to execute upon higher frequency cores. In addition, to lower context switching overhead without stifling the CPU migrations necessary to improve performance, the cache miss/reference ratio and the number of context switches and CPU migrations created by a running process will be used as scheduling criteria.

The objective of this research is to implement the approaches mentioned earlier to schedule the system's CPU resources, and, in the scenario where processes are assigned static or dynamic priorities that are not default priorities, to do so without creating a large context switching overhead nor adversely impacting process completion order as dictated by process priority.

4.2 Related Work

The work by Ghiasi, Keller, and Rawson [14] seems the closest to our own. The authors used a task-to-frequency scheduler that placed tasks in a multiprocessor system consisting of heterogeneous processors, each running at fixed but differing frequencies. However, their scheme risked an initial performance loss, especially in a system with many active tasks. They used an initialization phase that used insertion sort to place tasks in ready queues. This leads to a worst case scenario when all tasks are originally allocated to the fastest frequency. In contrast, our algorithm uses a “throughput estimate” that considers the number of processes in a core's ready queue, thus avoiding a situation in which most, if not all, processes are assigned to the same core.

Also, the authors in [14] used mechanisms for throttling the pipeline, or fetch throttling, to mimic the effects of frequency scaling. In contrast, our method uses kernel-directed frequency scaling to directly frequency scale cores. This is a much more direct, accurate, and effective means to scale and measure the frequencies of individual cores, and is not an approximation to frequency scaling. Our work also tests a wider range of core frequencies than [14]. In addition, we tested a task with eight concurrently executing processes, with each process having different execution characteristics. The authors in [14] did not perform a detailed analysis of a loaded system. Also, the effects, on performance, of how processors are paired were not examined.

In [35], Vajda presented a frequency scaling based scheduling algorithm. The algorithm allocated each core to only one application, however, and adjusted the frequency of the scalable cores, instead of using fixed frequency cores. This algorithm requires the operating system to actively re-calculate power budgets every time an application requests frequency boosting, which can result in performance loss. Also, unlike our work, the author relied on a simulator to evaluate his algorithm, rather than direct evaluation of his technique on real hardware.

Kumar, et al., [20] used a heterogeneity conscious thread-to-core assignment to demonstrate power savings. However, they also relied upon a simulator to measure their experimental results and to roughly model their cores. Also, there were minor differences in the instruction set architecture between the processors they modeled, since each processor was of a different generation. In contrast, our algorithm was executed upon a real multicore chip with identical cores, only differing in operating frequency. In addition, the authors assumed a maximum of one thread running at a time on only one core, and did not perform an analysis of a multithreaded application.

In another study [19], the same authors modeled a different set of cores to investigate processor power dissipation. However, again they used a simulator to model a diverse set of cores, rather than using real hardware. They also assumed that unused cores were completely powered down, rather than left idle, which is typically not the case in a real multicore system. Also, the authors constrained their problem to a single application switching among cores, and did not consider multiple threads on a single die.

Fedorova, Vengerov, and Doucette [11] presented a thread scheduler that attempts to balance core assignment with the goal of reducing completion time jitter and inconsistent priority enforcement while targeting optimal performance. However, this scheduler can generate increased context-switching overhead when attempting to allocate each thread's execution time evenly across all system's cores. Their algorithm is based upon reinforcement learning that allows each system core to learn a benefit function that approximates the future instruction rate on that core. This benefit function is then used to make core assignment decisions. In contrast, the algorithm presented in this paper uses feedback in which an application gives runtime information concerning both its instructions per cycle (IPC) and fraction wait time (time spent in ready and wait queues) to the operating system, which then uses this information to make scheduling decisions.

Almeida, et al., [2] used the difference between the desired and obtained throughput of a system to scale up or down the frequency of a processor. They repeated the process until the obtained throughput gradually became closer to the desired throughput. They used frequency scaling, in contrast to our method, which uses cores of different fixed frequencies. However, the default CPU frequency scaling policy found in most Linux based systems can also scale CPUs' frequencies, offering a similar, if not better, advantage than the method presented in [2]. Also, the algorithm presented by these authors only considers a system's throughput, and not an application's runtime IPC, which can be crucial in making scheduling decisions.

Isci, et al., [17] presented a global power manager that attempted to meet a specific global power budget by adjusting the power modes of individual cores. They assigned one of three power modes to each core, Turbo, Efficient1, and Efficient2. However, in contrast to our work, the authors used a simulator to evaluate their work, rather than real hardware. Also, they assumed a per-core DVFS (dynamic voltage and frequency scaling) knob to be available to their global power manager, unlike our work, which uses identical cores operating at different but fixed frequencies.

In [1] the authors proposed an algorithm in which all tasks were also assigned static priorities. In their model, tasks were assigned permanently to processors (partitioned scheduling) and were assigned rate-monotonic priorities that were inversely proportional to their periods. They measured the efficiency of their algorithm in terms of both total energy consumption and feasibility. They proved that this problem is NP-Hard in the strong sense on m $\geq$ 2 processors, even when feasibility is guaranteed *a priori.* However, unlike our algorithm, they used DVS (dynamic voltage scaling) to scale the frequency of their CPUs and used a simulator to measure their results.

Similar to our work, Schonherr, et al., [28] also proposed an approach that uses differing process priorities. In their approach, they assigned background (unimportant) processes a lower priority (nice value) than foreground (important) processes. They used a “separation in time” technique, in which foreground processes were allowed to complete before processor frequency was reduced, after which background processes were then allowed to execute. However, their approach only creates a power savings in some test cases, and not all test cases, as ours does. Also, they measured how much energy the execution of their load needed above idle consumption and not total energy consumption. Unlike our approach, their approach assumed that *a priori* knowledge, about the behavior and characteristics of running tasks, was available and was not determined at runtime.

4.3 Problem Specification

The goal of this research can be defined by four problem statements.

Specification 4.3.1 (Problem 1)**.** *Given a set S of processes P_i in a given task T to be executed by a heterogeneous multicore/manycore processor system M_i, with all $P_i^\prime s$ having identical priorities (the default Linux nice value of 0), schedule each P_i to execute upon the CPU(s) in M_i such that the global power budget of M_i is minimized, yet the performance gain of all $P_i^\prime s$ executing upon M_i is maximized, and the performance loss of all $P_i^\prime s$ executing upon M_i is minimized.*

Specification 4.3.2 (Problem 2)**.** *Given a set S of processes P_i in a given task T to be executed by a heterogeneous multicore/manycore processor system M_i, with each P_i having different static or dynamic (not necessarily unique) priorities, $nice_i$ (that may or may not* *be the default Linux nice value of 0), schedule each P_i to execute upon the CPU(s) in M_i such that the global power budget of M_i is minimized, yet the performance gain of all $P_i^{\prime}s$ executing upon M_i is maximized, and the performance loss of all $P_i^{\prime}s$ executing upon M_i is minimized.*

Specification 4.3.3 (Problem 3)**.** *Given a set S of processes P_i in a given task T to be executed by a heterogeneous multicore/manycore processor system M_i, with each P_i having different static or dynamic (not necessarily unique) priorities, $nice_i$ (that may or may not be the default Linux nice value of 0), schedule each P_i to execute upon the CPU(s) in M_i, while utilizing the cache miss/reference ratio of P_i at runtime as a scheduling factor, such that the global power budget of M_i is minimized, yet the performance gain of all $P_i^{\prime}s$ executing upon M_i is maximized, and the performance loss of all $P_i^{\prime}s$ executing upon M_i is minimized.*

Specification 4.3.4 (Problem 4)**.** *Given a set S of processes P_i in a given task T to be executed by a heterogeneous multicore/manycore processor system M_i, with each P_i having different static or dynamic (not necessarily unique) priorities, $nice_i$ (that may or may not be the default Linux nice value of 0), schedule each P_i to execute upon the CPU(s) in M_i, while utilizing the number of context switches and CPU migrations created by P_i at runtime as a scheduling factor, such that the global power budget of M_i is minimized, yet the performance gain of all $P_i^{\prime}s$ executing upon M_i is maximized, and the performance loss of all $P_i^{\prime}s$ executing upon M_i is minimized.*

CHAPTER 5

ALGORITHM 1

Algorithm *CPU Scheduler*

In this section we present an application-driven feedback mediated CPU scheduling algorithm, referred to as Algorithm *CPU Scheduler*. The algorithm does not use process priority as a variable.

5.1 Variables and Constants Used in Algorithm *CPU Scheduler*.

Algorithm *CPU Scheduler* uses the following variables and constants:

P_i :: *a single or multithreaded process*;

T :: *a task comprised of one or more processes* (P_i) *to be scheduled and executed by a multicore or manycore chip*;

num_cpus :: *number of CPUs* (*cores*) *on multicore or manycore chip*;

$core_i$ :: *a single core* (*CPU*) *on a multicore or manycore chip*;

$cpuset_i$ :: *a Linux cpuset*, *comprised of one or more homogeneous* $core_i's$, *all operating at the same frequency*; *however*, $core_i$ *and* $core_j$ *from a separate* $cpuset_i$ *or* $cpuset_j$ *may operate at different frequencies*;

$core\_freq_i$ :: *operating frequency of any* $core_i$ *in* $cpuset_i$ (*MHz*);

$ready\_queue_i$ :: *ready queue of any core in* $cpuset_i$;

$avg\_ready\_queue\_threads_i$ :: *average number of threads in* $ready\_queue_i$;

$throughput\_estimate\_a_i$ :: *estimate of the throughput of* $cpuset_i$;

```
    if ($avg\_ready\_queue\_threads_i \geq 1$)
      $throughput\_estimate\_a_i$
        $= avg\_ready\_queue\_threads_i * (1 \ / \ core\_freq_i)$;
    else
```

```
                $throughput\_estimate\_a_i = 1 \; / \; core\_freq_i;$
$throughput\_estimate\_b_i :: estimate\ of\ the\ inverse\ of\ throughput\ of\ cpuset_i;$
    $if\ (avg\_ready\_queue\_threads_i \geq 1)$
      $throughput\_estimate\_b_i$
        $= avg\_ready\_queue\_threads_i * core\_freq_i;$
          $else$
            $throughput\_estimate\_b_i = 1 \; / \; core\_freq_i;$
$num\_threads_i :: number\ of\ threads\ in\ process\ P_i;$
$IPC_i :: instructions\ per\ cycle\ for\ process\ P_i;$
    $IPC_i = \sum_{j=0}^{num\_threads_i} IPC_j \; / \; num\_threads_i,$
        $where\ j\ is\ a\ thread\ in\ P_i;$
$system\_time_i :: time\ spent\ by\ P_i\ while\ executing\ at\ the\ system\ level\ (kernel);$
$user\_time_i :: time\ spent\ by\ P_i\ while\ executing\ at\ the\ user\ level\ (application);$
$real\_time_i :: system\_time_i + user\_time_i;$
$fraction\_wait\_time_i :: fraction\ of\ total\ execution\ time\ spent\ by\ process\ P_i,$
                         $waiting\ in\ the\ ready\ queues\ and\ I/O\ queues;$
  $fraction\_wait\_time_i$
  $= \sum_{j=0}^{num\_threads_i} ((real\_time_j - (system\_time_j + user\_time_j))/real\_time_j)$
    $/ \; num\_threads_i,$
    $where\ j\ is\ a\ thread\ in\ P_i;$
$performance\_index_i :: a\ Boolean\ value\ calculated\ for\ an\ executing\ process\ P_i$
                         $from\ IPC_i\ and\ fraction\_wait\_time_i;$
  $performance\_index_i = true\ iff\ IPC_i < 1.7\ and\ fraction\_wait\_time_i > 0.5;$
$CPU\_intensity_i :: measure\ of\ the\ CPU\ intensity\ of\ process\ P_i;\ average\ CPU$
                    $utilization\ of\ all\ cores\ upon\ which\ P_i\ is\ executing$
$load\_average :: average\ system\ load\ over\ a\ period\ of\ time;\ number\ of\ processes$
                  $using\ or\ waiting\ for\ all\ CPUs\ (all\ core_i's);$
```

5.2 Description of Algorithm *CPU Scheduler*.

A description of the algorithm is as follows:

1. Processes are assigned to the ready queues of CPUs in a cpuset using a "throughput estimate" defined by the following two equations:

   If (avg. # of threads in ready queues of a cpuset ≥ 1)

   (a) *throughput_estimate_a* = avg. # of threads in that cpuset's ready queues * (1 / frequency of any core in that cpuset)

   (b) *throughput_estimate_b* = avg. # of threads in that cpuset's ready queues * frequency of any core in that cpuset

   (In case of a tie, assign process to cpuset with least average number of threads in ready queues.)

   Else

   (a) *throughput_estimate_a* = 1 / frequency of any core in that cpuset

   (b) *throughput_estimate_b* = 1 / frequency of any core in that cpuset

   Note: In Linux, there is no direct measurement of the number of processes in the ready queue of a single core (CPU) on a multicore chip. However, if processes are allocated to certain cpusets, an estimate of the average number of threads in the ready queue of a particular cpuset's core can be obtained by calculating ((run queue size of all ready queues combined / number of CPUs where CPU utilization = 100%) x average CPU utilization of CPU in that cpuset).

2. All newly-arriving processes are assigned to the cpuset with the least value of *throughput_estimate_*a_i. (If possible, all newly-arriving processes are assigned to the cpusets with CPUs operating at higher frequencies and with fewer processes in their ready queues).

3. As a process P_i executes, a "performance index" is calculated from the IPC (instructions per cycle) and fraction wait time of P_i. This calculation is made asynchronously with respect to other concurrently executing processes.

4. If IPC_i is lower than a threshold (< 1.7), or $fraction\_wait\_time_i$ is higher than a threshold (> 0.5), then the performance index for that process, $performance\_index_i$ is evaluated as true. (The performance index of a process is evaluated as true if the process has decreased performance, caused by a decrease in its instructions per cycle rate, or due to longer periods of time spent by the process in the ready or wait queues. We chose an IPC threshold of 1.7 based upon process performance data indicating that these processes had an IPC value greater than 1.7 during initial phases of execution. We chose a fraction wait time threshold of 0.5 because it is the average of 0, which signifies that a process is not spending any time waiting in the ready and I/O queues, and 1, which signifies that a process is spending all of its execution time waiting in the ready and I/O queues).

5. At regular time intervals, for an executing process, $performance\_index_i$ is calculated. Each interval is equal to approximately one-fifth of the process's total execution time.

   (a) If the process is CPU intensive (the average CPU utilization of all cores upon which it is executing is $> 50\%$) and if the performance index is true for that process, $throughput\_estimate\_a_i$ is recalculated for all cpusets, and the process is context switched to the cpuset with the lowest value of $throughput\_estimate\_a_i$. (A cpuset with a lower value of $throughput\_estimate\_a_i$ will contain CPUs operating at higher frequencies and fewer processes in the ready queues of its cores. The goal of context switching a CPU intensive process that has decreased performance to such a cpuset is to speed up the execution of that process. We chose a CPU utilization threshold of 50% to categorize a process as CPU intensive because we found, in our experiments, that the non-CPU intensive benchmarks used for our test cases all had a CPU utilization less than 50%, and the CPU intensive benchmarks that we tested all had a CPU utilization greater than

50%).

(b) To account for performance saturation caused by memory access, if a process is not CPU intensive and if the load average is $> num\_cpus$, then the process is context switched to the cpuset with the lowest value of $throughput\_estimate\_b_i$. (If the system is heavily loaded and the process is not CPU intensive, the process is context switched to a cpuset with lower frequency cores). If the load average is $< num\_cpus$ and if the performance index is true, then this process is context switched to the cpuset with the lowest value of $throughput\_estimate\_a_i$. (If the system is not heavily loaded, and the process has decreased performance, then the process is context-switched to a cpuset with higher frequency cores).

6. For a non-executing process, if the fraction wait time of that process, $fraction\_wait\_time_i$, is equal to 1, then $throughput\_estimate\_a_i$ is recalculated for all cpusets, and the process is assigned to the cpuset with the lowest value of $throughput\_estimate\_a_i$. (If the process is waiting in the ready queues for the cores of a cpuset for too long, it is context switched to a cpuset with fewer processes in its ready queues).

7. At regular time intervals, the average number of threads in the ready queues of all cpusets is computed. If the average number of threads in the ready queues of any cpuset is less than 1, then a process is assigned to that cpuset. If two or more cpusets have an average number of threads in their ready queues that is less than 1, a process is first assigned to the cpuset with CPUs operating at the highest frequency. The process(s) chosen to migrate to the new cpuset are chosen in order of CPU intensity, with higher CPU intensive processes having higher priority. (If there is a cpuset containing a core with an empty ready queue, then a process is assigned to this core).

8. Thread parallelism is accounted for by the number and frequency of CPUs in cpusets (hardware), with the lack of parallelism available in cpusets with fewer CPUs compensated for by the higher frequencies of its CPUs.

9. The measure of the algorithm's effectiveness is:

    (a) Average Instructions Per Second / Watts

    $= (\sum_{i=0}^{m} (Instructions\ Executed\ by\ P_i\ /\ Execution\ Time\ (seconds)$

    $for\ P_i)$ / m) / watts,

    where m = number of processes in task T

    (b) Average time required for all processes in task T to complete execution * watts

    $= (\sum_{i=0}^{m} Execution\ Time\ (seconds)\ for\ P_i$ / m) * watts,

    where m = number of processes in Task T

    (A potentially more efficient algorithm will have a higher value of (a) and a lower value of (b))

$Algorithm\ 1 : Feedback - driven\ CPU\ Scheduling\ Algorithm$

$(CPU\ Scheduler)$

```
Start with a set $S$ of processes $P_i$ in a given task $T$:
  for all $P_i$ in $T$ (concurrently){
    //initially schedule a process
    calculate $throughput\_estimate\_a_i$ for all cpusets;
    assign $P_i$ to cpuset with least value of $throughput\_estimate\_a_i$;
    //if non − executing process is in wait queue too long, context switch process
    if($fraction\_wait\_time_i == 1$){
      recalculate $throughput\_estimate\_a_i$ for all cpusets and assign $P_i$ to cpuset
      with least value of $throughput\_estimate\_a_i$;
    }
    do{
      execute $P_i$;
      calculate $CPU\_intensity_i$ for $P_i$;
      calculate $IPC_i$ and $fraction\_wait\_time_i$ for $P_i$;
      compute $performance\_index_i$ of $P_i$ using $IPC_i$ and $fraction\_wait\_time_i$;
      measure $load\_average$;
      if($CPU\_intensity_i > 50\%$ and $performance\_index_i == true$){
          recalculate $throughput\_estimate\_a_i$ for all cpusets and contextswitch
          $P_i$ to cpuset with least value of $throughput\_estimate\_a_i$;
       }
       else if($CPU\_intensity_i < 50\%$ and $load\_average > num\_cpus$){
           calculate $throughput\_estimate\_b_i$ for all cpusets and context switch $P_i$
           to cpuset with least value of $throughput\_estimate\_b_i$;
         }
         else if($CPU\_intensity_i < 50\%$ and $load\_average < num\_cpus$ and
             $performance\_index_i == true$){
               recalculate $throughput\_estimate\_a_i$ for all cpusets and context
               switch $P_i$ to cpuset with least value of $throughput\_estimate\_a_i$;
            }
      calculate $CPU\_intensity_i$ for $P_i$;

      //if a ready queue is empty, then assign a process to it
      if(a ready queue is empty and $P_i$ is process with highest value of
         $CPU\_intensity_i$ of all executing processes){
         contextswitch $P_i$ to cpuset with the empty ready queue;
         }
   }until $P_i$ terminates;
}
```

CHAPTER 6

ALGORITHM 2

Algorithm *Priority CPU Scheduler*

In this section we present an application-driven feedback mediated CPU scheduling algorithm that utilizes static or dynamic process priorities, referred to as Algorithm *Priority CPU Scheduler*. This algorithm uses a process's priority (nice value) as a variable. Thus, unlike Algorithm *CPU Scheduler* , process priority is used as one of the determining factors to schedule processes.

6.1 Variables and Constants Used in Algorithm *Priority CPU Scheduler*.

Algorithm *Priority CPU Scheduler* uses the following variables and constants:

P_i :: *a single or multithreaded process*;

T :: *a task comprised of one or more processes* (P_i) *to be scheduled and executed by a multicore or manycore chip*;

num_cpus :: *number of CPUs* (*cores*) *on multicore or manycore chip*;

$core_i$ :: *a single core* (*CPU*) *on a multicore or manycore chip*;

$cpuset_i$ :: *a Linux cpuset*, *comprised of one or more homogeneous* $core_i's$, *all operating at the same frequency*; *however*, $core_i$ *and* $core_j$ *from a separate* $cpuset_i$ *or* $cpuset_j$ *may operate at different frequencies*;

$core\_freq_i$ :: *operating frequency of any* $core_i$ *in* $cpuset_i$ (MHz);

$ready\_queue_i$ :: *readyqueue of any core in* $cpuset_i$;

$avg\_ready\_queue\_threads_i$ :: *average number of threads in* $ready\_queue_i$;

$throughput\_estimate\_a_i$ :: *estimate of the throughput of* $cpuset_i$;

```
    $if(avg\_ready\_queue\_threads_i \geq 1)$
      $throughput\_estimate\_a_i$
```

```
            $= avg\_ready\_queue\_threads_i * (1/\ core\_freq_i);$
             $else$
                 $throughput\_estimate\_a_i = 1\ /\ core\_freq_i;$
$throughput\_estimate\_b_i :: estimate\ of\ the\ inverse\ of\ throughput\ of\ cpuset_i;$
        $if\ (avg\_ready\_queue\_threads_i \geq 1)$
          $throughput\_estimate\_b_i$
            $= avg\_ready\_queue\_threads_i * core\_freq_i;$
             $else$
                 $throughput\_estimate\_b_i = 1\ /\ core\_freq_i;$
$num\_threads_i :: number\ of\ threads\ in\ process\ P_i;$
$IPC_i :: instructions\ per\ cycle\ for\ process\ P_i;$
        $IPC_i = \sum_{j=0}^{num\_threads_i} IPC_j\ /\ num\_threads_i,$
            $where\ j\ is\ a\ thread\ in\ P_i;$
$system\_time_i :: time\ spent\ by\ P_i\ while\ executing\ at\ the\ system\ level\ (kernel);$
$user\_time_i :: time\ spent\ by\ P_i\ while\ executing\ at\ the\ user\ level\ (application);$
$real\_time_i :: system\_time_i + user\_time_i;$
$fraction\_wait\_time_i :: fraction\ of\ total\ execution\ time\ spent\ by\ process\ P_i,$
                              $waiting\ in\ the\ ready\ queues\ and\ I/O\ queues;$
   $fraction\_wait\_time_i$
   $= \sum_{j=0}^{num\_threads_i} ((real\_time_j - (system\_time_j + user\_time_j))/real\_time_j)$
      $/\ num\_threads_i,$
      $where\ j\ is\ a\ thread\ in\ P_i;$
$performance\_index_i :: a\ Boolean\ value\ calculated\ for\ an\ executing\ process\ P_i$
                              $from\ IPC_i\ and\ fraction\_wait\_time_i;$
   $performance\_index_i = true\ iff\ IPC_i < 1.7\ and\ fraction\_wait\_time_i > 0.5;$
$CPU\_intensity_i :: measure\ of\ the\ CPU\ intensity\ of\ process\ P_i;\ average\ CPU$
                         $utilization\ of\ all\ cores\ upon\ which\ P_i\ is\ executing$
$nice_i :: the\ priority\ of\ process\ P_i;\ may\ be\ a\ value\ between\ -20\ and\ 19;$
          $a\ lower\ value\ indicates\ a\ higher\ priority;$
```

$CPU\_intensity\_priority\_index_i$ :: *an index that takes into account the CPU intensity and priority of process* P_i

$$CPU\_intensity\_priority\_index_i = (0.5 * CPU\_intensity_i) + (0.5 * \frac{1}{nice_i});$$

$load\_average$ :: *average system load over a period of time*; *number of processes using or waiting for all CPUs* (*all core$_i$′s*);

6.2 Description of Algorithm *Priority CPU Scheduler*.

A description of the algorithm is as follows:

1. All processes are assigned a nice value, which signifies the priority of that process. This nice value may be between -20 and 19, with a lower value indicating a higher priority.

2. Processes are assigned to the ready queues of CPUs in a cpuset using a "throughput estimate" defined by the following two equations:

   If (avg. # of threads in ready queues of a cpuset $\geq$ 1)

   (a) *throughput_estimate_a* = avg. # of threads in that cpuset's ready queues * (1 / frequency of any core in that cpuset)

   (b) *throughput_estimate_b* = avg. # of threads in that cpuset's ready queues * frequency of any core in that cpuset

   (In case of a tie, assign process to cpuset with least average number of threads in ready queues.)

   Else

   (a) *throughput_estimate_a* = 1 / frequency of any core in that cpuset

   (b) *throughput_estimate_b* = 1 / frequency of any core in that cpuset

Note: In Linux, there is no direct measurement of the number of processes in the ready queue of a single core (CPU) on a multicore chip. However, if processes are allocated to certain cpusets, an estimate of the average number of threads in the ready queue of a particular cpuset's core can be obtained by calculating ((run queue size of all ready queues combined / number of CPUs where CPU utilization = 100%) x average CPU utilization of CPU in that cpuset).

3. All newly-arriving processes are assigned to the cpuset with the least value of $throughput\_estimate\_a_i$. (If possible, all newly-arriving processes are assigned to the cpusets with CPUs operating at higher frequencies and with fewer processes in their ready queues).

4. As a process P_i executes, a "performance index" is calculated from the IPC (instructions per cycle) and fraction wait time of P_i. This calculation is made asynchronously with respect to other concurrently executing processes.

5. If IPC_i is lower than a threshold (< 1.7), or $fraction\_wait\_time_i$ is higher than a threshold (> 0.5), then the performance index for that process, $performance\_index_i$ is evaluated as true. (The performance index of a process is evaluated as true if the process has decreased performance, caused by a decrease in its instructions per cycle rate, or due to longer periods of time spent by the process in the ready or wait queues. We chose an IPC threshold of 1.7 based upon process performance data indicating that these processes had an IPC value greater than 1.7 during initial phases of execution. We chose a fraction wait time threshold of 0.5 because it is the average of 0, which signifies that a process is not spending any time waiting in the ready and I/O queues, and 1, which signifies that a process is spending all of its execution time waiting in the ready and I/O queues).

6. At regular time intervals, for an executing process, $performance\_index_i$ is calculated. Each interval is equal to approximately one-fifth of the process's total execution time.

    (a) If the process is CPU intensive (the average CPU utilization of all cores upon which it is executing is $> 50\%$, its nice value is < 9, and if the performance index is true for that process, *throughput_estimate_*a_i is recalculated for all cpusets, and the process is context switched to the cpuset with the lowest value of *throughput_estimate_*a_i. (A cpuset with a lower value of *throughput_estimate_*a_i will contain CPUs operating at higher frequencies and fewer processes in the ready queues of its cores. The goal of context switching a high priority, CPU intensive process that has decreased performance to such a cpuset is to speed up the execution of that process. We chose a CPU utilization threshold of 50% to categorize a process as CPU intensive because we found, in our experiments, that the non-CPU intensive benchmarks used for our cases all had a CPU utilization less than 50%, and the test CPU intensive benchmarks that we tested all had a CPU utilization greater than 50%.)

(b) To account for performance saturation caused by memory access, if a process is <u>not</u> CPU intensive, its nice value is $>$ -1, and if the load average is $>$ *num_cpus*, or if a process is not CPU intensive, its nice value is $\leq$ -1, and the load average is $>$ 2 * *num_cpus*, then the process is context switched to the cpuset with the lowest value of *throughput_estimate_*b_i. (If the system is heavily loaded and the process is not CPU intensive, the process is context switched to a cpuset with lower frequency cores. The higher the priority of the process, the more heavily loaded the system must be before that process is context switched.) If the non-CPU intensive process's nice value is $\leq$ -11, the load average is $\leq$ *num_cpus*, and if the performance index is true, or if the process's nice value is $>$ -11, the load average is $<$ 0.15 * *num_cpus*, and if the performance index is true, then this process is context switched to the cpuset with the lowest value of *throughput_estimate_*a_i. (If the system is not heavily loaded, and the process has decreased performance, then the process is context-switched to a cpuset with higher frequency cores. The lower the priority of the process, the lower the load average must be before the process is context switched.)

7. For a non-executing process, if the fraction wait time of that process, $fraction\_wait\_time_i$, is $\geq$ 0.7 and its nice value is < -1, or if $fraction\_wait\_time_i$ of that process is > 0.9, then $throughput\_estimate\_a_i$ is recalculated for all cpusets, and the process is assigned to the cpuset with the lowest value of $throughput\_estimate\_a_i$. (If the process is waiting in the ready queues for the cores of a cpuset for too long, it is context switched to a cpuset with fewer processes in its ready queues. The higher the priority of the process, the less time it must spend waiting in the ready queues before it is context switched.)

8. For all executing processes, a "CPU_intensity_priority_index" is calculated from the CPU intensity and nice value of that process.

9. At regular time intervals, the average number of threads in the ready queues of all cpusets is computed. If the average number of threads in the ready queues of any cpuset is < 1, then a process is assigned to that cpuset. If two or more cpusets have an average number of threads in their ready queues that is < 1, a process is first assigned to the cpuset with CPUs operating at the highest frequency. Process(s) are chosen to migrate to the new cpuset according to their CPU_intensity_priority_index, with those having a higher value of this index having higher priority. (If there is a cpuset containing a core with an empty ready queue, then a process is assigned to this core. Process(s) are chosen according to both their CPU intensity and process priority.)

10. Thread parallelism is accounted for by the number and frequency of CPUs in cpusets (hardware), with the lack of parallelism available in cpusets with fewer CPUs compensated for by the higher frequencies of its CPUs.

11. The measure of the algorithm's effectiveness is:

    (a) Average Instructions Per Second / Watts

    $= (\sum_{i=0}^{m} (Instructions\ Executed\ by\ P_i\ /\ Execution\ Time\ (seconds)\ for\ P_i)$ / m) / watts,

    where m = number of processes in task T

(b) Average time required for all processes in task T to complete execution * watts

$= (\sum_{i=0}^{m} Execution\ Time\ (seconds)\ for\ P_i$ / m) * watts,

where m = number of processes in Task T

(A potentially more efficient algorithm will have a higher value of (a) and a lower value of (b))

Algorithm 2 : Feedback − driven Priority − based CPU Scheduling Algorithm (*Priority CPU Scheduler*)

```
Start with a set $S$ of processes $P_i$ in a given task $T$:
  for all $P_i$ in $T$ (concurrently){
    //initially schedule a process
    calculate $throughput\_estimate\_a_i$ for all cpusets;
    assign $P_i$ to cpuset with least value of $throughput\_estimate\_a_i$;
    //if non − executing process is in wait queue too long, context switch process
    if($fraction\_wait\_time_i \geq 0.7$ and $nice_i < -1$){
      recalculate $throughput\_estimate\_a_i$ for all cpusets and assign $P_i$ to cpuset
      with least value of $throughput\_estimate\_a_i$;
    }
    else if($fraction\_wait\_time_i > 0.9$){
      recalculate $throughput\_estimate\_a_i$ for all cpusets and assign $P_i$ to cpuset
      with least value of $throughput\_estimate\_a_i$;
    }
    do{
      execute $P_i$;
      calculate $CPU\_intensity_i$ for $P_i$;
      calculate $IPC_i$ and $fraction\_wait\_time_i$ for $P_i$;
      compute $performance\_index_i$ of $P_i$ using $IPC_i$ and $fraction\_wait\_time_i$;
      measure $load\_average$;
      if($CPU\_intensity_i > 50\%$ and $nice_i < 9$ and $performance\_index_i == true$){
          recalculate $throughput\_estimate\_a_i$ for all cpusets and context switch
          $P_i$ to cpuset with least value of $throughput\_estimate\_a_i$;
      }
      else if($CPU\_intensity_i < 50\%$ and $nice_i > -1$ and $load\_average > num\_cpus$){
          calculate $throughput\_estimate\_b_i$ for all cpusets and context switch $P_i$
```

```
            $to\ cpuset\ with\ least\ value\ of\ throughput\_estimate\_b_i$;
          }
          $else\ if(CPU\_intensity_i < 50\%\ and\ nice_i \leq -1\ and\ load\_average$
                $> 2 * num\_cpus$){
                 $calculate\ throughput\_estimate\_b_i\ for\ all\ cpusets\ and\ context$
                 $switch\ P_i\ to\ cpuset\ with\ least\ value\ of\ throughput\_estimate\_b_i$;
           }
           $else\ if(CPU\_intensity_i < 50\%\ and\ nice_i \leq -11\ and\ load\_average$
                 $\leq num\_cpus\ and\ performance\_index_i == true$){
                  $recalculate\ throughput\_estimate\_a_i\ for\ all\ cpusets\ and\ context$
                  $switch\ P_i\ to\ cpuset\ with\ least\ value\ of\ throughput\_estimate\_a_i$;
             }
            $else\ if(CPU\_intensity_i < 50\%\ and\ nice_i > -11\ and\ load\_average$
                  $< 0.15 * num\_cpus\ and\ performance\_index_i == true$){
                   $recalculate\ throughput\_estimate\_a_i\ for\ all\ cpusets\ and\ context$
                   $switch\ P_i\ to\ cpuset\ with\ least\ value\ of\ throughput\_estimate\_a_i$;
              }
      $compute\ CPU\_intensity\_priority\_index_i\ for\ P_i$;
      $//if\ a\ ready\ queue\ is\ empty,\ then\ assign\ a\ process\ to\ it$
      $if(a\ ready\ queue\ is\ empty\ and\ P_i\ is\ process\ with\ highest\ value\ of$
        $CPU\_intensity\_priority\_index_i\ of\ all\ executing\ processes$){
        $context\ switch\ P_i\ to\ cpuset\ with\ the\ empty\ ready\ queue$;
        }
  }$until\ P_i\ terminates$;
}
```

CHAPTER 7

ALGORITHM 3

Algorithm *Cache Miss Priority CPU Scheduler*

In this section we present an application-driven feedback mediated CPU scheduling algorithm that considers the ratio between the total number of cache misses and the total number of cache references generated at runtime. It also utilizes static or dynamic process priorities, and is referred to as Algorithm *Cache Miss Priority CPU Scheduler*.

7.1 Variables and Constants Used in Algorithm *Cache Miss Priority CPU Scheduler*.

Algorithm *Cache Miss Priority CPU Scheduler* uses the following variables and constants:

P_i :: *a single or multithreaded process*;

T :: *a task comprised of one or more processes* (P_i) *to be scheduled and executed by a multicore or manycore chip*;

num_cpus :: *number of CPUs* (*cores*) *on multicore or manycore chip*;

$core_i$ :: *a single core* (*CPU*) *on a multicore or manycore chip*;

$cpuset_i$ :: *a Linux cpuset*, *comprised of one or more homogeneous* $core_i's$, *all operating at the same frequency*; *however*, $core_i$ *and* $core_j$ *from a separate* $cpuset_i$ *or* $cpuset_j$ *may operate at different frequencies*;

$core\_freq_i$ :: *operating frequency of any* $core_i$ *in* $cpuset_i$ (*MHz*);

$ready\_queue_i$ :: *ready queue of any core in* $cpuset_i$;

$avg\_ready\_queue\_threads_i$ :: *average number of threads in* $ready\_queue_i$;

$throughput\_estimate\_a_i$ :: *estimate of the throughput of* $cpuset_i$;

```
    if ($avg\_ready\_queue\_threads_i \geq 1$)
      $throughput\_estimate\_a_i$
```

```
            $= avg\_ready\_queue\_threads_i * (1 \ / \ core\_freq_i);$
             $else$
                $throughput\_estimate\_a_i = 1 \ / \ core\_freq_i;$
$throughput\_estimate\_b_i :: estimate\ of\ the\ inverse\ of\ throughput\ of\ cpuset_i;$
      $if\ (avg\_ready\_queue\_threads_i \geq 1)$
        $throughput\_estimate\_b_i$
            $= avg\_ready\_queue\_threads_i * core\_freq_i;$
             $else$
                $throughput\_estimate\_b_i = 1 \ / \ core\_freq_i;$
$num\_threads_i :: number\ of\ threads\ in\ process\ P_i;$
$IPC_i :: instructions\ per\ cycle\ for\ process\ P_i;$
      $IPC_i = \sum_{j=0}^{num\_threads_i} IPC_j \ / \ num\_threads_i,$
            $where\ j\ is\ a\ thread\ in\ P_i;$
$system\_time_i :: time\ spent\ by\ P_i\ while\ executing\ at\ the\ system\ level\ (kernel);$
$user\_time_i :: time\ spent\ by\ P_i\ while\ executing\ at\ the\ user\ level\ (application);$
$real\_time_i :: system\_time_i + user\_time_i;$
$fraction\_wait\_time_i :: fraction\ of\ total\ execution\ time\ spent\ by\ process\ P_i,$
                          $waiting\ in\ the\ ready\ queues\ and\ I/O\ queues;$
  $fraction\_wait\_time_i$
  $= \sum_{j=0}^{num\_threads_i} ((real\_time_j - (system\_time_j + user\_time_j))/real\_time_j)$
    $/ \ num\_threads_i,$
    $where\ j\ is\ a\ thread\ in\ P_i;$
$performance\_index_i :: a\ Boolean\ value\ calculated\ for\ an\ executing\ process\ P_i$
                          $from\ IPC_i\ and\ fraction\_wait\_time_i;$
  $performance\_index_i = true\ iff\ IPC_i < 1.7\ and\ fraction\_wait\_time_i > 0.5;$
$cache\_miss_i :: the\ number\ of\ cache\ misses\ generated\ by\ process\ P_i;$
$cache\_reference_i :: the\ number\ of\ cache\ references\ generated\ by\ process\ P_i;$
$cache\_miss\_index_i :: a\ Boolean\ value\ calculated\ for\ an\ executing\ process\ P_i\ that$
                        $uses\ the\ ratio\ between\ cache\_miss_i\ and\ cache\_reference_i;$
```

$cache\_miss\_index_i = true\ iff\ cache\_miss_i\ /\ cache\_reference_i > 0.0014;$

$CPU\_intensity_i :: measure\ of\ the\ CPU\ intensity\ of\ process\ P_i;\ average\ CPU$
$utilization\ of\ all\ cores\ upon\ which\ P_i\ is\ executing$

$nice_i :: the\ priority\ of\ process\ P_i;\ may\ be\ a\ value\ between\ -20\ and\ 19;$
$a\ lower\ value\ indicates\ a\ higher\ priority;$

$CPU\_intensity\_priority\_index_i :: an\ index\ that\ takes\ into\ account\ the\ CPU$
$intensity\ and\ priority\ of\ process\ P_i$

$$CPU\_intensity\_priority\_index_i = (0.5 * CPU\_intensity_i) + (0.5 * \frac{1}{nice_i});$$

$load\_average :: average\ system\ load\ over\ a\ period\ of\ time;\ number\ of\ processes$
$using\ or\ waiting\ for\ all\ CPUs\ (all\ core_i's);$

7.2 Description of Algorithm *Cache Miss Priority CPU Scheduler*.

A description of the algorithm is as follows:

1. All processes are assigned a nice value, which signifies the priority of that process. This nice value may be between -20 and 19, with a lower value indicating a higher priority.

2. Processes are assigned to the ready queues of CPUs in a cpuset using a "throughput estimate" defined by the following two equations:

   If (avg. # of threads in ready queues of a cpuset ≥ 1)

   (a) *throughput_estimate_a* = avg. # of threads in that cpuset's ready queues * (1 / frequency of any core in that cpuset)

   (b) *throughput_estimate_b* = avg. # of threads in that cpuset's ready queues * frequency of any core in that cpuset

   (In case of a tie, assign process to cpuset with least average number of threads in ready queues.)

   Else

(a) $throughput\_estimate\_a$ = 1 / frequency of any core in that cpuset

(b) $throughput\_estimate\_b$ = 1 / frequency of any core in that cpuset

Note: In Linux, there is no direct measurement of the number of processes in the ready queue of a single core (CPU) on a multicore chip. However, if processes are allocated to certain cpusets, an estimate of the average number of threads in the ready queue of a particular cpuset's core can be obtained by calculating ((run queue size of all ready queues combined / number of CPUs where CPU utilization = 100%) x average CPU utilization of CPU in that cpuset).

3. All newly-arriving processes are assigned to the cpuset with the least value of $throughput\_estimate\_a_i$. (If possible, all newly-arriving processes are assigned to the cpusets with CPUs operating at higher frequencies and with fewer processes in their ready queues).

4. As a process P_i executes, a "performance index" is calculated from the IPC (instructions per cycle) and fraction wait time of P_i. This calculation is made asynchronously with respect to other concurrently executing processes.

5. As a process P_i executes, a "cache miss index" is also calculated from the number of cache misses and the number of cache references generated by P_i. This calculation is made asynchronously with respect to other concurrently executing processes.

6. If IPC_i is lower than a threshold ($<$ 1.7), or $fraction\_wait\_time_i$ is higher than a threshold ($>$ 0.5), then the performance index for that process, $performance\_index_i$ is evaluated as true. (The performance index of a process is evaluated as true if the process has decreased performance, caused by a decrease in its instructions per cycle rate, or due to longer periods of time spent by the process in the ready or wait queues. We chose an IPC threshold of 1.7 based upon process performance data indicating that these processes had an IPC value greater than 1.7 during initial phases of execution. We chose a fraction wait time threshold of 0.5 because it is the average of 0, which signifies that a process is not spending any time

waiting in the ready and I/O queues, and 1, which signifies that a process is spending all of its execution time waiting in the ready and I/O queues).

7. If $cache\_miss_i$ / $cache\_reference_i$ is higher than a threshold (> 0.0014), then the cache miss index for that process, $cache\_miss\_index_i$, is evaluated as true. (The cache miss index of a process is evaluated as true if the process generates a large number of cache misses with respect to the number of cache references generated. We chose a cache miss index threshold of 0.0014 because we wanted to use a threshold that would force our algorithm to alter its scheduling strategy if the ratio between the number of cache misses and cache references was greater than half of the average value for the cache miss index of all executing processes in our experimental group. Based upon experimental data, a value of 0.0014 represents this threshold.)

8. At regular time intervals, for an executing process, $performance\_index_i$ and $cache\_miss\_index_i$ are calculated. Each interval is equal to approximately one-fifth of the process's total execution time.

   (a) If the process is CPU intensive (the average CPU utilization of all cores upon which it is executing is $> 50\%$, its nice value is < 9, the performance index is true for that process, and if the cache miss index is false for that process, $throughput\_estimate\_a_i$ is recalculated for all cpusets, and the process is context switched to the cpuset with the lowest value of $throughput\_estimate\_a_i$. (A cpuset with a lower value of $throughput\_estimate\_a_i$ will contain CPUs operating at higher frequencies and fewer processes in the ready queues of its cores. The goal of context switching a high priority, CPU intensive process that does not generate many cache misses, and that has decreased performance, to such a cpuset is to speed up the execution of that process. We chose a CPU utilization threshold of 50% to categorize a process as CPU intensive because we found, in our experiments, that the non-CPU intensive benchmarks used for our cases all had a CPU utilization less than 50%, and the test CPU intensive benchmarks that we tested all had a CPU utilization greater than 50%.)

(b) To account for performance saturation caused by memory access, if a process is not CPU intensive, its nice value is $>$ -1, and if the load average is $>$ *num_cpus*, or if a process is not CPU intensive, its nice value is $\leq$ -1, and the load average is $>$ 2 * *num_cpus*, then the process is context switched to the cpuset with the lowest value of $throughput\_estimate\_b_i$. (If the system is heavily loaded and the process is not CPU intensive, the process is context switched to a cpuset with lower frequency cores. The higher the priority of the process, the more heavily loaded the system must be before that process is context switched.) If the non-CPU intensive process's nice value is $\leq$ -11, the load average is $\leq$ *num_cpus*, the performance index for that process is true, and if the cache miss index for that process is false, or if the process's nice value is $>$ -11, the load average is $<$ 0.15 * *num_cpus*, the performance index for that process is true, and if the cache miss index for that process is false, then this process is context switched to the cpuset with the lowest value of $throughput\_estimate\_a_i$. (If the system is not heavily loaded, and a process does not generate a large number of cache misses and has decreased performance, then the process is context-switched to a cpuset with higher frequency cores. The lower the priority of the process, the lower the load average must be before the process is context switched.)

9. For a non-executing process, if the fraction wait time of that process, $fraction\_wait\_time_i$, is ≥ 0.7 and its nice value is $<$ -1, or if $fraction\_wait\_time_i$ of that process is > 0.9, then $throughput\_estimate\_a_i$ is recalculated for all cpusets, and the process is assigned to the cpuset with the lowest value of $throughput\_estimate\_a_i$. (If the process is waiting in the ready queues for the cores of a cpuset for too long, it is context switched to a cpuset with fewer processes in its ready queues. The higher the priority of the process, the less time it must spend waiting in the ready queues before it is context switched.)

10. For all executing processes, a "CPU_intensity_priority_index" is calculated from the CPU intensity and nice value of that process.

11. At regular time intervals, the average number of threads in the ready queues of all cpusets is computed. If the average number of threads in the ready queues of any cpuset is < 1, then a process is assigned to that cpuset. If two or more cpusets have an average number of threads in their ready queues that is < 1, a process is first assigned to the cpuset with CPUs operating at the highest frequency. Process(s) are chosen to migrate to the new cpuset according to their CPU_intensity_priority_index, with those having a higher value of this index having higher priority. (If there is a cpuset containing a core with an empty ready queue, then a process is assigned to this core. Process(s) are chosen according to both their CPU intensity and process priority.)

12. Thread parallelism is accounted for by the number and frequency of CPUs in cpusets (hardware), with the lack of parallelism available in cpusets with fewer CPUs compensated for by the higher frequencies of its CPUs.

13. The measure of the algorithm's effectiveness is:

    (a) Average Instructions Per Second / Watts

    $= (\sum_{i=0}^{m} (Instructions\ Executed\ by\ P_i\ /\ Execution\ Time\ (seconds)$

    $for\ P_i)$ / m) / watts,

    where m = number of processes in task T

    (b) Average time required for all processes in task T to complete execution * watts

    $= (\sum_{i=0}^{m} Execution\ Time\ (seconds)\ for\ P_i$ / m) * watts,

    where m = number of processes in Task T

    (A potentially more efficient algorithm will have a higher value of (a) and a lower value of (b))

Algorithm 3 : Cache Miss Feedback − driven Priority − based CPU Scheduling Algorithm (*Cache Miss Priority CPU Scheduler*)

```
Start with a set $S$ of processes $P_i$ in a given task $T$:
 for all $P_i$ in $T$ (concurrently){
   //initially schedule a process
   calculate $throughput\_estimate\_a_i$ for all cpusets;
   assign $P_i$ to cpuset with least value of $throughput\_estimate\_a_i$;
   //if non − executing process is in wait queue too long, context switch process
   if($fraction\_wait\_time_i \geq 0.7$ and $nice_i < -1$){
     recalculate $throughput\_estimate\_a_i$ for all cpusets and assign $P_i$ to cpuset
     with least value of $throughput\_estimate\_a_i$;
   }
   else if($fraction\_wait\_time_i > 0.9$){
     recalculate $throughput\_estimate\_a_i$ for all cpusets and assign $P_i$ to cpuset
     with least value of $throughput\_estimate\_a_i$;
   }
   do{
     execute $P_i$;
     calculate $CPU\_intensity_i$ for $P_i$;
     calculate $IPC_i$ and $fraction\_wait\_time_i$ for $P_i$;
     compute $performance\_index_i$ of $P_i$ using $IPC_i$ and $fraction\_wait\_time_i$;
     compute $cache\_miss\_index_i$ of $P_i$ using $cache\_miss_i$ and $cache\_reference_i$;
     measure $load\_average$;
     if($CPU\_intensity_i > 50\%$ and $nice_i < 9$ and $performance\_index_i == true$
       and $cache\_miss\_index_i == false$){
         recalculate $throughput\_estimate\_a_i$ for all cpusets and context switch
         $P_i$ to cpuset with least value of $throughput\_estimate\_a_i$;
      }
```

```
            else if($CPU\_intensity_i < 50\%$ and $nice_i > -1$ and $load\_average > num\_cpus$){
                calculate $throughput\_estimate\_b_i$ for all cpusets and context switch $P_i$
                to cpuset with least value of $throughput\_estimate\_b_i$;
              }
             else if($CPU\_intensity_i < 50\%$ and $nice_i \leq -1$ and $load\_average$
                   $> 2 * num\_cpus$){
                    calculate $throughput\_estimate\_b_i$ for all cpusets and context
                    switch $P_i$ to cpuset with least value of $throughput\_estimate\_b_i$;
               }
              else if($CPU\_intensity_i < 50\%$ and $nice_i \leq -11$ and $load\_average$
                    $\leq num\_cpus$ and $performance\_index_i == true$ and
                    $cache\_miss\_index_i == false$){
                      recalculate $throughput\_estimate\_a_i$ for all cpusets and context
                      switch $P_i$ to cpuset with least value of $throughput\_estimate\_a_i$;
                 }
               else if($CPU\_intensity_i < 50\%$ and $nice_i > -11$ and $load\_average$
                     $< 0.15 * num\_cpus$ and $performance\_index_i == true$ and
                     $cache\_miss\_index_i == false$){
                       recalculate $throughput\_estimate\_a_i$ for all cpusets and context
                       switch $P_i$ to cpuset with least value of $throughput\_estimate\_a_i$;
                   }
          compute $CPU\_intensity\_priority\_index_i$ for $P_i$;
          //if a ready queue is empty, then assign a process to it
          if(a ready queue is empty and $P_i$ is process with highest value of
             $CPU\_intensity\_priority\_index_i$ of all executing processes){
             context switch $P_i$ to cpuset with the empty ready queue;
             }
     }until $P_i$ terminates;
   }
```

CHAPTER 8

ALGORITHM 4

Algorithm *Context Switch Priority CPU Scheduler*

In this section we present an application-driven feedback mediated CPU scheduling algorithm that considers the number of context switches and the number of CPU migrations generated at runtime. It also utilizes static or dynamic process priorities, and is referred to as Algorithm *Context Switch Priority CPU Scheduler*.

8.1 Variables and Constants Used in Algorithm *Context Switch Priority CPU Scheduler*

.

Algorithm *Context Switch Priority CPU Scheduler* uses the following variables and constants:

P_i :: *a single or multithreaded process*;

T :: *a task comprised of one or more processes* (P_i) *to be scheduled and executed by a multicore or manycore chip*;

num_cpus :: *number of CPUs* (*cores*) *on multicore or manycore chip*;

$core_i$ :: *a single core* (CPU) *on a multicore or manycore chip*;

$cpuset_i$ :: *a Linux cpuset*, *comprised of one or more homogeneous* $core_i's$, *all operating at the same frequency*; *however*, $core_i$ *and* $core_j$ *from a separate* $cpuset_i$ *or* $cpuset_j$ *may operate at different frequencies*;

$core\_freq_i$ :: *operating frequency of any* $core_i$ *in* $cpuset_i$ (MHz);

$ready\_queue_i$ :: *ready queue of any core in* $cpuset_i$;

$avg\_ready\_queue\_threads_i$ :: *average number of threads in* $ready\_queue_i$;

$throughput\_estimate\_a_i$ :: *estimate of the throughput of* $cpuset_i$;

$if(avg\_ready\_queue\_threads_i \geq 1)$

```
    $throughput\_estimate\_a_i$
      $= avg\_ready\_queue\_threads_i * (1 / core\_freq_i);$
       $else$
          $throughput\_estimate\_a_i = 1 / core\_freq_i;$
$throughput\_estimate\_b_i$ :: $estimate\ of\ the\ inverse\ of\ throughput\ of\ cpuset_i;$
  $if\ (avg\_ready\_queue\_threads_i \geq 1)$
    $throughput\_estimate\_b_i$
      $= avg\_ready\_queue\_threads_i * core\_freq_i;$
       $else$
          $throughput\_estimate\_b_i = 1 / core\_freq_i;$
$num\_threads_i$ :: $number\ of\ threads\ in\ process\ P_i;$
$IPC_i$ :: $instructions\ per\ cycle\ for\ process\ P_i;$
  $IPC_i = \sum_{j=0}^{num\_threads_i} IPC_j / num\_threads_i,$
      $where\ j\ is\ a\ thread\ in\ P_i;$
$system\_time_i$ :: $time\ spent\ by\ P_i\ while\ executing\ at\ the\ system\ level\ (kernel);$
$user\_time_i$ :: $time\ spent\ by\ P_i\ while\ executing\ at\ the\ user\ level\ (application);$
$real\_time_i$ :: $system\_time_i + user\_time_i;$
$fraction\_wait\_time_i$ :: $fraction\ of\ total\ execution\ time\ spent\ by\ process\ P_i,$
                    $waiting\ in\ the\ ready\ queues\ and\ I/O\ queues;$
 $fraction\_wait\_time_i$
 $= \sum_{j=0}^{num\_threads_i} ((real\_time_j - (system\_time_j + user\_time_j))/real\_time_j)$
   $/ num\_threads_i,$
   $where\ j\ is\ a\ thread\ in\ P_i;$
$performance\_index_i$ :: $a\ Boolean\ value\ calculated\ for\ an\ executing\ process\ P_i$
                    $from\ IPC_i\ and\ fraction\_wait\_time_i;$
 $performance\_index_i = true\ iff\ IPC_i < 1.7\ and\ fraction\_wait\_time_i > 0.5;$
$context\_switch_i$ :: $the\ number\ of\ context\ switches\ generated\ by\ process\ P_i;$
$CPU\_migration_i$ :: $the\ number\ of\ CPU\ migrations\ generated\ by\ process\ P_i;$
$context\_switch\_CPU\_migration\_index_i$ :: $a\ Boolean\ value\ calculated\ for\ an$
```

$executing\ process\ P_i\ that\ uses$

$context\_switch_i\ and\ CPU\_migration_i;$

$$context\_switch\_CPU\_migration\_index_i = true\ iff\ (0.5 * \frac{context\_switch_i}{real\_time_i}) + (0.5 * \frac{CPU\_migration_i}{real\_time_i}) > 17.0;$$

$CPU\_intensity_i :: measure\ of\ the\ CPU\ intensity\ of\ process\ P_i;\ average\ CPU$

$utilization\ of\ all\ cores\ upon\ which\ P_i\ is\ executing$

$nice_i :: the\ priority\ of\ process\ P_i;\ may\ be\ a\ value\ between$ -20 and 19;

$a\ lower\ value\ indicates\ a\ higher\ priority;$

$CPU\_intensity\_priority\_index_i :: an\ index\ that\ takes\ into\ account\ the\ CPU$

$ntensity\ and\ priority\ of\ process\ P_i$

$$CPU\_intensity\_priority\_index_i = (0.5 * CPU\_intensity_i) + (0.5 * \frac{1}{nice_i});$$

$load\_average :: average\ system\ load\ over\ a\ period\ of\ time;\ number\ of\ processes$

$using\ or\ waiting\ for\ all\ CPUs\ (all\ core_i's);$

8.2 Description of Algorithm *Context Switch Priority CPU Scheduler*.

A description of the algorithm is as follows:

1. All processes are assigned a nice value, which signifies the priority of that process. This nice value may be between -20 and 19, with a lower value indicating a higher priority.

2. Processes are assigned to the ready queues of CPUs in a cpuset using a "throughput estimate" defined by the following two equations:

   If (avg. # of threads in ready queues of a cpuset $\geq$ 1)

   (a) *throughput_estimate_a* = avg. # of threads in that cpuset's ready queues * (1 / frequency of any core in that cpuset)

   (b) *throughput_estimate_b* = avg. # of threads in that cpuset's ready queues * frequency of any core in that cpuset

(In case of a tie, assign process to cpuset with least average number of threads in ready queues.)

Else

(a) $throughput\_estimate\_a$ = 1 / frequency of any core in that cpuset

(b) $throughput\_estimate\_b$ = 1 / frequency of any core in that cpuset

Note: In Linux, there is no direct measurement of the number of processes in the ready queue of a single core (CPU) on a multicore chip. However, if processes are allocated to certain cpusets, an estimate of the average number of threads in the ready queue of a particular cpuset's core can be obtained by calculating ((run queue size of all ready queues combined / number of CPUs where CPU utilization = 100%) x average CPU utilization of CPU in that cpuset).

3. All newly-arriving processes are assigned to the cpuset with the least value of $throughput\_estimate\_a_i$. (If possible, all newly-arriving processes are assigned to the cpusets with CPUs operating at higher frequencies and with fewer processes in their ready queues).

4. As a process P_i executes, a "performance index" is calculated from the IPC (instructions per cycle) and fraction wait time of P_i. This calculation is made asynchronously with respect to other concurrently executing processes.

5. As a process P_i executes, a "context switch CPU migration index" is also calculated from the number of context switches per second of execution time (user time + system time) and the number of CPU migrations per second of execution time (user time + system time) generated by P_i. This calculation is made asynchronously with respect to other concurrently executing processes.

6. If IPC_i is lower than a threshold (< 1.7), or $fraction\_wait\_time_i$ is higher than a threshold (> 0.5), then the performance index for that process, $performance\_index_i$ is evaluated as true. (The performance index of a process is

evaluated as true if the process has decreased performance, caused by a decrease in its instructions per cycle rate, or due to longer periods of time spent by the process in the ready or wait queues. We chose an IPC threshold of 1.7 based upon process performance data indicating that these processes had an IPC value greater than 1.7 during initial phases of execution. We chose a fraction wait time threshold of 0.5 because it is the average of 0, which signifies that a process is not spending any time waiting in the ready and I/O queues, and 1, which signifies that a process is spending all of its execution time waiting in the ready and I/O queues).

7. If $(0.5 * \frac{context\_switch_i}{real\_time_i}) + (0.5 * \frac{CPU\_migration_i}{real\_time_i})$ is higher than a threshold (> 17.0), then the context switch CPU migration index for that process, $context\_switch\_CPU\_migration\_index_i$, is evaluated as true. (The context switch CPU migration index of a process is evaluated as true if the process generates a large number of context switches per second of execution time and CPU migrations per second of execution time. We chose a context switch CPU migration index threshold of 17.0 because we wanted to use a threshold that would force our algorithm to alter its scheduling strategy if the context switch CPU migration index of a process was greater than half of the average value for the context switch CPU migration index of all executing processes in our experimental group. Based upon experimental data, a value of 17.0 represents this threshold.)

8. At regular time intervals, for an executing process, $performance\_index_i$ and $context\_switch\_CPU\_migration\_index_i$ are calculated. Each interval is equal to approximately one-fifth of the process's total execution time.

   (a) If the process is CPU intensive (the average CPU utilization of all cores upon which it is executing is $> 50\%$, its nice value is < 9, the performance index is true for that process, and if the context switch CPU migration index is false for that process, $throughput\_estimate\_a_i$ is recalculated for all cpusets, and the process is context switched to the cpuset with the lowest value of $throughput\_estimate\_a_i$. (A cpuset with a lower value of $throughput\_estimate\_a_i$ will contain CPUs operating at higher frequencies and fewer processes in the ready queues of its cores. The goal of context switching a high priority, CPU intensive process that does not undergo many context switches and CPU migrations, and that has decreased performance, to such a cpuset is to speed up the execution of that process. We chose a CPU utilization threshold of 50% to categorize a process as CPU intensive because we found, in our experiments, that the non-CPU intensive benchmarks used for our cases all had a CPU utilization less than 50%, and the test CPU intensive benchmarks that we tested all had a CPU utilization greater than 50%.)

(b) To account for performance saturation caused by memory access, if a process is <u>not</u> CPU intensive, its nice value is > -1, and if the load average is $> num\_cpus$, or if a process is not CPU intensive, its nice value is $\leq$ -1, and the load average is > 2 * num_cpus, then the process is context switched to the cpuset with the lowest value of $throughput\_estimate\_b_i$. (If the system is heavily loaded and the process is not CPU intensive, the process is context switched to a cpuset with lower frequency cores. The higher the priority of the process, the more heavily loaded the system must be before that process is context switched.) If the non-CPU intensive process's nice value is $\leq$ -11, the load average is $\leq num\_cpus$, the performance index for that process is true, and if the context switch CPU migration index for that process is false, or if the process's nice value is > -11, the load average is < 0.15 * num_cpus, the performance index for that process is true, and if the context switch CPU migration index for that process is false, then this process is context switched to the cpuset with the lowest value of $throughput\_estimate\_a_i$. (If the system is not heavily loaded, and a process does not undergo a large number of context switches and CPU migrations per second of execution time and has decreased performance, then the process is context-switched to a cpuset with higher frequency cores. The lower the priority of the process, the lower the load average must be before the process is context switched.)

9. For a non-executing process, if the fraction wait time of that process, $fraction\_wait\_time_i$, is $\geq$ 0.7 and its nice value is < -1, or if $fraction\_wait\_time_i$ of that process is > 0.9, then $throughput\_estimate\_a_i$ is recalculated for all cpusets, and the process is assigned to the cpuset with the lowest value of $throughput\_estimate\_a_i$. (If the process is waiting in the ready queues for the cores of a cpuset for too long, it is context switched to a cpuset with fewer processes in its ready queues. The higher the priority of the process, the less time it must spend waiting in the ready queues before it is context switched.)

10. For all executing processes, a "CPU_intensity_priority_index" is calculated from the CPU intensity and nice value of that process.

11. At regular time intervals, the average number of threads in the ready queues of all cpusets is computed. If the average number of threads in the ready queues of any cpuset is < 1, then a process is assigned to that cpuset. If two or more cpusets have an average number of threads in their ready queues that is < 1, a process is first assigned to the cpuset with CPUs operating at the highest frequency. Process(s) are chosen to migrate to the new cpuset according to their CPU_intensity_priority_index, with those having a higher value of this index having higher priority. (If there is a cpuset containing a core with an empty ready queue, then a process is assigned to this core. Process(s) are chosen according to both their CPU intensity and process priority.)

12. Thread parallelism is accounted for by the number and frequency of CPUs in cpusets (hardware), with the lack of parallelism available in cpusets with fewer CPUs compensated for by the higher frequencies of its CPUs.

13. The measure of the algorithm's effectiveness is:

    (a) Average Instructions Per Second / Watts

    $= (\sum_{i=0}^{m} (Instructions\ Executed\ by\ P_i\ /\ Execution\ Time\ (seconds)\ for\ P_i)$ / m) / watts,

    where m = number of processes in task T

(b) Average time required for all processes in task T to
complete execution * watts
$= (\sum_{i=0}^{m} Execution\ Time\ (seconds)\ for\ P_i$ / m) * watts,
where m = number of processes in Task T

(A potentially more efficient algorithm will have a higher value of (a) and a lower value of (b))

Algorithm 4 : Context Switch Feedback − driven Priority − based CPU Scheduling Algorithm

(Context Switch Priority CPU Scheduler)

```
Start with a set S of processes $P_i$ in a given task T:
 for all $P_i$ in T (concurrently){
   //initially schedule a process
   calculate $throughput\_estimate\_a_i$ for all cpusets;
   assign $P_i$ to cpuset with least value of $throughput\_estimate\_a_i$;
   //if non − executing process is in wait queue too long, context switch process
   if($fraction\_wait\_time_i \geq 0.7$ and $nice_i < -1$){
     recalculate $throughput\_estimate\_a_i$ for all cpusets and assign $P_i$ to cpuset
     with least value of $throughput\_estimate\_a_i$;
   }
   else if($fraction\_wait\_time_i > 0.9$){
     recalculate $throughput\_estimate\_a_i$ for all cpusets and assign $P_i$ to cpuset
     with least value of $throughput\_estimate\_a_i$;
   }
   do{
     execute $P_i$;
     calculate $CPU\_intensity_i$ for $P_i$;
     calculate $IPC_i$ and $fraction\_wait\_time_i$ for $P_i$;
     compute $performance\_index_i$ of $P_i$ using $IPC_i$ and $fraction\_wait\_time_i$;
     compute $context\_switch\_CPU\_migration\_index_i$ of $P_i$ using $context\_switch_i$
             and $CPU\_migration_i$;
     measure $load\_average$;
     if($CPU\_intensity_i > 50\%$ and $nice_i < 9$ and $performance\_index_i == true$
       and $context\_switch\_CPU\_migration\_index_i == false$){
         recalculate $throughput\_estimate\_a_i$ for all cpusets and context switch
```

```
                    $P_i$ to cpuset with least value of $throughput\_estimate\_a_i$;
                }
                else if($CPU\_intensity_i < 50\%$ and $nice_i > -1$ and $load\_average > num\_cpus$){
                        calculate $throughput\_estimate\_b_i$ for all cpusets and context switch $P_i$
                        to cpuset with least value of $throughput\_estimate\_b_i$;
                  }
                  else if($CPU\_intensity_i < 50\%$ and $nice_i \leq -1$ and $load\_average$
                             $> 2 * num\_cpus$){
                              calculate $throughput\_estimate\_b_i$ for all cpusets and context
                              switch $P_i$ to cpuset with least value of $throughput\_estimate\_b_i$;
                    }
                    else if($CPU\_intensity_i < 50\%$ and $nice_i \leq -11$ and $load\_average$
                              $\leq num\_cpus$ and $performance\_index_i == true$ and
                              $context\_switch\_CPU\_migration\_index_i == false$){
                                recalculate $throughput\_estimate\_a_i$ for all cpusets and context
                                switch $P_i$ to cpuset with least value of $throughput\_estimate\_a_i$;
                      }
                      else if($CPU\_intensity_i < 50\%$ and $nice_i > -11$ and $load\_average$
                                $< 0.15 * num\_cpus$ and $performance\_index_i == true$ and
                                $context\_switch\_CPU\_migration\_index_i == false$){
                                  recalculate $throughput\_estimate\_a_i$ for all cpusets and context
                                  switch $P_i$ to cpuset with least value of $throughput\_estimate\_a_i$;
                        }
            compute $CPU\_intensity\_priority\_index_i$ for $P_i$;
            //if a ready queue is empty, then assign a process to it
            if(a ready queue is empty and $P_i$ is process with highest value of
                $CPU\_intensity\_priority\_index_i$ of all executing processes){
                context switch $P_i$ to cpuset with the empty ready queue;
                }
        }until $P_i$ terminates;
    }
```

CHAPTER 9

EVALUATION AND RESULTS

Algorithms 1, 2, 3, and 4 utilize "application-driven" feedback to dynamically context switch processes in a given task upon heterogeneous cpusets. Each algorithm considers different performance metrics and thus uses different variables at runtime to achieve power savings while maximizing performance gains and minimizing performance losses.

9.1 Methodology

This section discusses the experimental setup and various methodologies used in this research, including hardware and operating system, CPU frequency scaling, creation of cpusets, measurement of application performance, context switching of processes, and measurement of power usage.

Hardware and operating system. We ran our experiments on a system equipped with an 8 core 2.3 GHz AMD Opteron 6134 "MagnyCours" processor [4]. This is a 64 bit multicore processor that has eight identical cores and supports per core frequency scaling.

The operating system we used was the Fedora Core 14 Linux distribution with a version 2.6.35 kernel.

CPU frequency scaling. To create a heterogeneous architecture, we frequency scaled each core using the Linux "cpufreq" kernel module. We changed the default Linux CPU frequency scaling governor, which is the "on demand" governor, to the "userspace" governor. This allowed us to then adjust the CPU frequencies for the individual cores. The default Linux "on demand" governor sets the CPU frequency based upon the current usage of the CPUs, in terms of CPU utilization, and the system load.

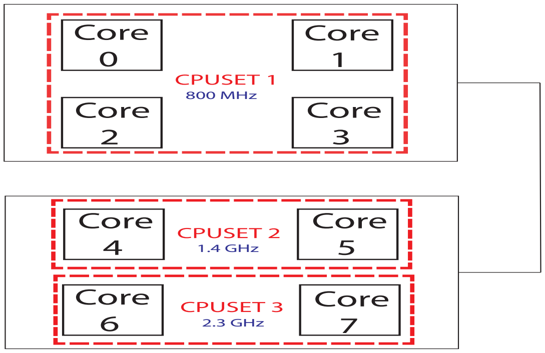


Figure 7: Hardware Partitioning

Creation of cpusets. We partitioned the cores into hardware partitions using "CPUSETS for Linux". We partitioned the eight cores into three cpusets, with each core in the same cpuset operating at the same frequency, but cores from different cpusets operating at different frequencies. The first cpuset contained four cores, each operating at 800 MHz. The second cpuset contained two cores, each operating at 1.4 GHz. The third cpuset contained two cores, each operating at 2.3 GHz. We used this strategy to compensate for the lack of parallelism available in cpusets with fewer CPUs. Figure 7 shows the hardware partitioning strategy we used.

Priority Assignment. For Algorithm 2 we assigned both static and dynamic priorities (Linux nice values) in the range of -20 to 19 to each executing benchmark using the set-priority() system call. For Algorithms 3 and 4 we assigned dynamic priorities within this same range using the same system call. When assigning static priorities, each process was assigned a nice value and retained its resulting priority throughout its five execution cycles. When assigning dynamic priorities, each process retained its first priority for its first three execution cycles, and was then assigned a different priority for its last two execution cycles.

Measurement of application performance. We used the standard "perf" hardware performance counters subsystem in Linux to obtain per core and per application performance statistics. These statistics were used at runtime to give feedback to the operating system concerning an application's execution characteristics and to gather performance data.

Context switching of applications. Applications were context switched between cpusets by the operating system, based upon feedback from that application, by setting the CPU affinity mask using the "sched_setaffinity()" system call.

Measurement of Power Usage. We directly measured the power usage of our system during execution using the "KillAWatt" meter by Arbor Scientific.

9.2 Discussion of Results

For our research, we wished to investigate the performance gains or losses and power savings achieved by Algorithms *CPU Scheduler*, *Priority CPU Scheduler*, *Cache Miss Priority CPU Scheduler*, and *Context Switch Priority CPU Scheduler* when executing a typical workload. To this end, we wanted to include a combination of both CPU intensive and non-CPU intensive applications in our workload. Hence we tested our algorithm using five different benchmarks, four from the SPEC CPU2006 suite and one non-CPU intensive benchmark that we wrote. The SPEC CPU2006 benchmarks, representing a variety of architectural characteristics, were bzip2, gcc, mcf, and sjeng. We also used a non-CPU intensive benchmark that we wrote in C.

We utilized four of the SPEC CPU2006 benchmarks because we wanted to test three, five, eight, and twenty-four concurrently executing benchmarks, while including a combination of both CPU intensive and non-CPU intensive benchmarks. We did this to ensure all three cpusets and all eight cores in our three cpuset, eight core evaluation platform had processes executing upon them, and to demonstrate how our algorithm behaves with a combination of both types of benchmarks. When we measured the CPU intensity of all the benchmarks in the SPEC CPU2006 suite, we found that all of them were CPU intensive, except for mcf. Thus, for three concurrently executing benchmarks, we chose two CPU intensive benchmarks, bzip2 and gcc, and one non-CPU intensive benchmark, noncpuintensive. We chose a similar combination for the five and eight concurrently executing benchmark test cases, and since most of the SPEC CPU2006 benchmarks are CPU intensive, we could only include a subset of them to ensure a combination of both types of benchmarks in our test cases. Also, when measuring the results for a heavily loaded system, we wanted

Task	Concurrently Executing Benchmarks
1	bzip2, gcc, noncpuintensive
2	bzip2, gcc, mcf, sjeng, noncpuintensive
3	bzip2, gcc, mcf, sjeng, noncpuintensive$_{(1-4)}$
4	bzip2$_{(1-3)}$, gcc$_{(1-3)}$, mcf$_{(1-3)}$, sjeng$_{(1-3)}$, noncpuintensive$_{(1-12)}$

Table 1: Benchmarks and Tasks Evaluated

to use the same benchmarks that we used in our test cases containing three, five, and eight concurrently executing processes, in order to eliminate as many experimental variables as possible when comparing a heavily loaded system to a lightly loaded system.

We ran four tasks, composed of the following processes. As shown in Table 1, Task 1 was composed of the bzip2, gcc, and noncpuintensive benchmarks. Task 2 was composed of the bzip2, gcc, mcf, sjeng, and noncpuintensive benchmarks. Task 3 was composed of the bzip2, gcc, mcf, sjeng, and four separate instances of the noncpuintensive benchmarks. Task 4 was composed of three separate instances of bzip2, three separate instances of gcc, three separate instances of mcf, three separate instances of sjeng, and twelve separate instances of noncpuintensive, for a total of twenty-four benchmarks. When a task was executed, the benchmarks within a task were run concurrently and were restarted once they finished until each benchmark completed execution five times. Each task was run separately.

We used two experimental groups, a test group in which a task was scheduled using either Algorithm *CPU Scheduler*, Algorithm *Priority CPU Scheduler*, Algorithm *Cache Miss Priority CPU Scheduler*, or Algorithm *Context Switch Priority CPU Scheduler*, and a control group in which the same task was scheduled using the default Linux scheduler and "on demand" CPU frequency scaling governor. Hence each task was executed twice. We chose the "on demand" governor for our control group because it is the default CPU frequency scaling governor in the Fedora Core 14 package, but more importantly, because we wanted to compare the effects of using our algorithm versus a governor that would employ a different approach to advantageously using CPU frequency scaling.

9.2.1 Evaluation of Algorithm *CPU Scheduler*

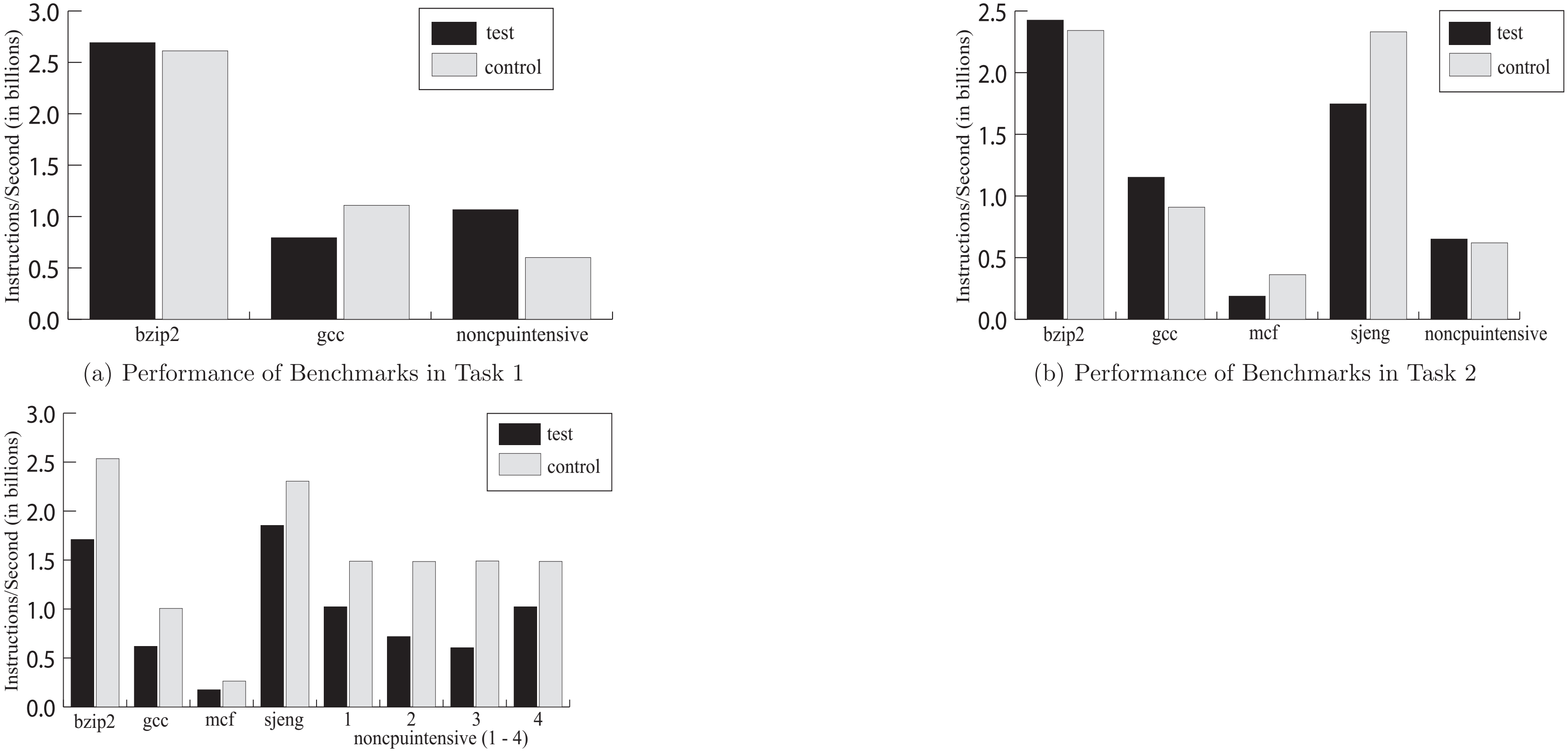
In our evaluation of Algorithm 1 (*CPU Scheduler*), we assumed that each process in the system had the same default Linux nice value of 0, and thus the same default priority. This algorithm did not use process priority as a variable. The results of this evaluation are summarized in Tables 2-4 and Figures 8-10 as follows.

We measured the completion times and instructions executed for each benchmark (process) in a task, and then calculated the instructions per second (I/S) for each process. For simplicity's sake, we only show these results for Tasks 1-3. These results are shown in Table 2 and Figure 8(a)-(c). For all four tasks, we then used the average of these values to calculate the instructions per second for the entire task. We also measured the power usage, in watts, that the system used while executing a particular task. We then used these values to calculate two metrics, (I/S)/watts and (average process completion time) · watts for each task in the test and control groups. Table 3 and Figure 9(a)-(c) depict these results for Tasks 1-4. Finally, for all four tasks, we calculated the performance gain/loss and power savings of the test versus control group, as well as the percent improvement of both metrics obtained by using Algorithm *CPU Scheduler* versus the default Linux scheduler and CPU frequency scaling governor. These results are presented in Table 4 and Figure 10(a) and Figure 10(b).

Instructions/Second (in billions)

	Task 1		Task 2		Task 3	
	test	control	test	control	test	control
bzip2	2.692	2.612	2.426	2.342	1.709	2.535
gcc	0.794	1.109	1.152	0.909	0.619	1.006
mcf	–	–	0.188	0.362	0.176	0.264
sjeng	–	–	1.747	2.331	1.854	2.305
noncpuintensive$_1$	1.067	0.601	0.651	0.620	1.022	1.488
noncpuintensive$_2$	–	–	–	–	0.719	1.485
noncpuintensive$_3$	–	–	–	–	0.604	1.490
noncpuintensive$_4$	–	–	–	–	1.022	1.486

Table 2: Performance of Benchmarks Evaluated for Algorithm 1



(a) Performance of Benchmarks in Task 1

(b) Performance of Benchmarks in Task 2

(c) Performance of Benchmarks in Task 3

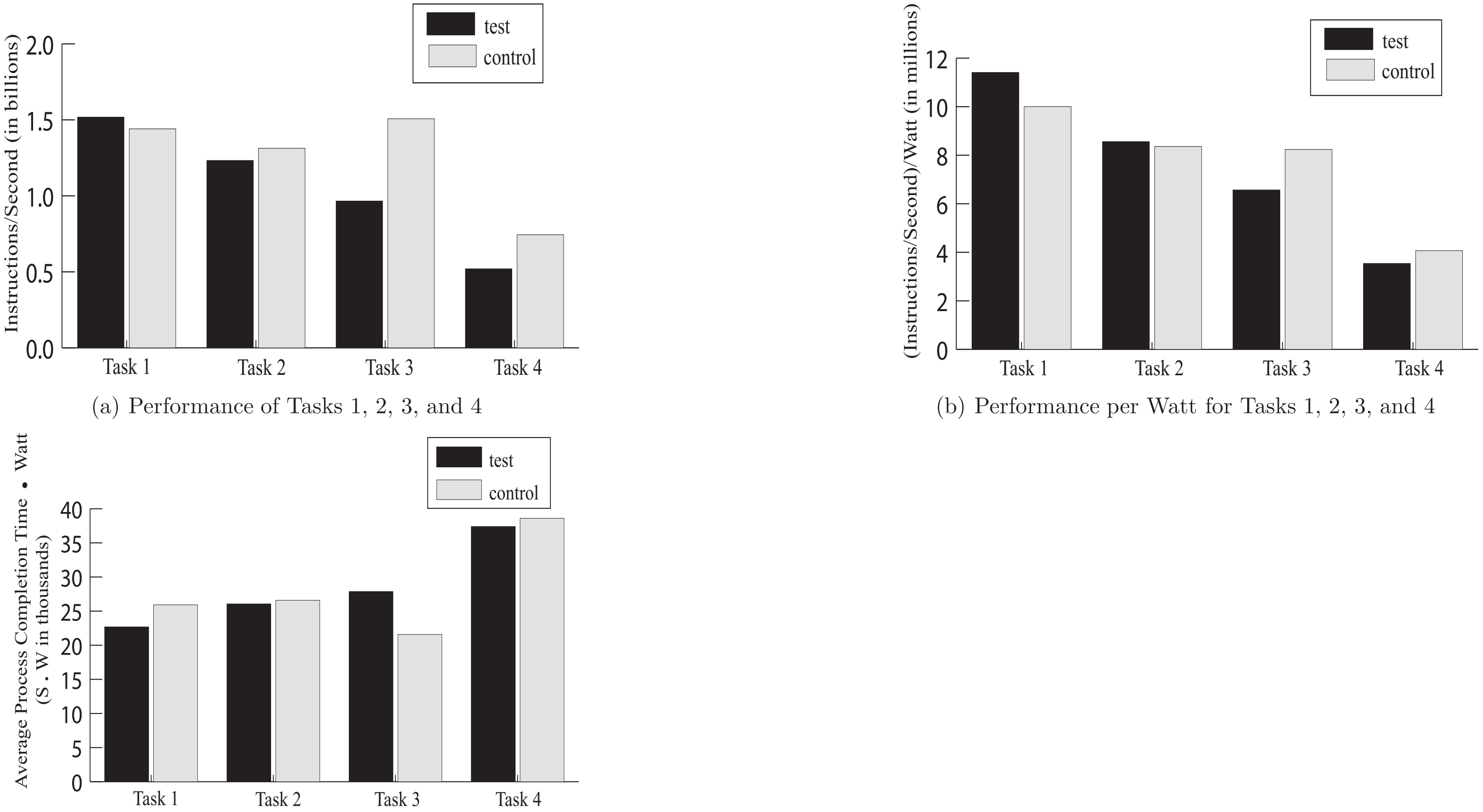
Figure 8: Performance of Benchmarks Evaluated for Algorithm 1

Table 2 and Figure 8 indicate that the higher CPU intensive processes bzip2, sjeng, and, to a lesser degree, gcc, have a higher instructions per second value than mcf and noncpuintensive, which are less CPU intensive and more memory intensive. We believe this is caused by two factors. Firstly, memory performance saturation did occur when executing memory intensive benchmarks, especially mcf. Thus the delay created by waiting for data from memory caused a decrease in the overall instructions per second rate for those processes. Secondly, the higher CPU intensive processes were being context switched to cpusets with higher frequency cores by Algorithm *CPU Scheduler*. Our algorithm context switches CPU intensive processes (whose performance index is true) to the higher frequency cpusets and, in a heavily loaded system, moves less CPU intensive processes to lower frequency cores. Also, if a higher frequency cpuset has cores with an empty ready queue, our algorithm gives priority to higher CPU intensive processes to be assigned to that cpuset. These two characteristics allow Algorithm *CPU Scheduler* to advantageously schedule processes with the goal of lowering the execution time and improving the performance of the entire task.

Also, as shown in Figure 8, for Task 1 and Task 2, the test group had a higher instructions per second value than the control group, for the majority of the benchmarks in that task. This may also be due to the fact that our algorithm allows processes to execute upon CPU cores that are better suited to their current execution characteristics, compared to the default Linux scheduler.

	Task 1		Task 2		Task 3		Task 4	
	test	control	test	control	test	control	test	control
Performance Avg Ins/Sec (in billions)	1.518	1.441	1.233	1.313	0.966	1.507	0.520	0.744
Performance per Watt Avg (Ins/Sec)/Watt (in millions	11.411	10.004	8.560	8.361	6.569	8.237	3.541	4.067
Execution Time · Watt Avg Time (Sec) · Watt (in thousands)	22.691	25.921	26.060	26.590	27.875	21.579	37.398	38.607

Table 3: Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4



(a) Performance of Tasks 1, 2, 3, and 4

(b) Performance per Watt for Tasks 1, 2, 3, and 4

(c) Execution Time · Watt for Tasks 1, 2, 3, and 4

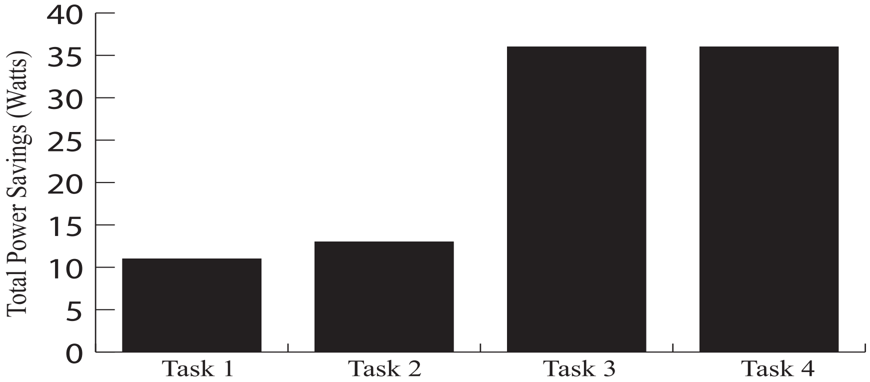
Figure 9: Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4

Table 3 and Figure 9(a) show that the test group had a higher performance value than the control group when executing Task1. This indicates that using our algorithm can produce a higher instructions per second rate for a task with three concurrently executing processes than the default Linux scheduler and “on demand” governor.

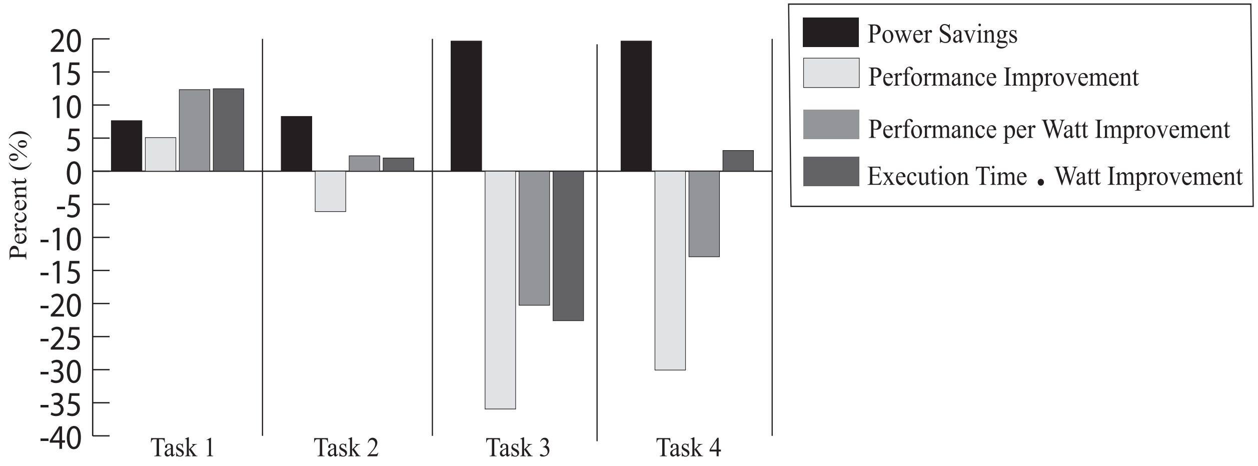
Also, as shown in Table 3 and Figure 9(b), the metric performance per watt for both Task 1 and Task 2 was higher for the test group than the control group. In addition, as indicated by Figure 9(c), the metric execution time · watt was lower for Task 1, Task 2, and Task 4 for the test group than the control group. This indicates that for a heavily loaded system (in this case one with twenty-four concurrently executing processes), our algorithm is better when taking into account both average process completion time and power consumption. Also, since a higher value for the first metric and a lower value for the second metric indicate a potentially more efficient algorithm, when taking into account both power savings and performance, we believe that for tasks with three or five concurrently executing processes, our algorithm is better than the default Linux scheduler and “on demand” CPU frequency scaling governor.

	Task 1	Task 2	Task 3	Task 4
Total Power Savings (Watts)	11	13	36	36
Power Savings (%)	7.64	8.28	19.67	19.67
Performance Improvement (%)	5.08	-6.10	-35.94	-30.06
Performance per Watt Improvement (%)	12.33	2.32	-20.26	-12.93
Execution Time · Watt Improvement (%)	12.46	1.99	-22.59	3.13

Table 4: Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4



(a) Total Power Savings for Tasks 1, 2, 3, and 4



(b) Percent Power Savings and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4

Figure 10: Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4

As indicated by Table 4 and Figure 10(a), the total power savings for Tasks 1, 2, 3, and 4 obtained by using Algorithm *CPU Scheduler* is 11 watts, 13 watts, 36 watts, and 36 watts, respectively. This shows that the power savings obtained by our algorithm is significant and becomes even more significant as the number of concurrently executing processes is greater. The percent power savings, relative to the control group, is also significant, as shown in Table 4 and Figure 10(b). It can be as high as 19.67%. For all four tasks, there is a significant increase in power savings obtained by our algorithm.

In addition, Table 4 and Figure 10(b) show that there is not only a power savings, but also a performance improvement, obtained by Algorithm *CPU Scheduler* when executing a task with three concurrently executing benchmarks. We believe this is due to the fact that in a lightly loaded system, there are more cores with empty ready queues. In this situation, our algorithm will exhibit better performance because it will context switch even a lower CPU intensive process to the higher CPU frequency cpusets, if it is the only process executing in the system or if the system is not loaded, whereas an "on demand" governor, or other dynamic CPU frequency scaling technique, will lower the core's frequency upon which that non-CPU intensive process is executing.

While there is a performance loss associated with using Algorithm *CPU Scheduler* for Task 2, Task 3, and Task 4, Table 4 and Figure 10(b) show that there is a positive improvement in the performance per watt and execution time · watt for Task 2, and a positive improvement in the execution time · watt for Task 4. Task 1 shows an improvement in performance, performance per watt, and execution time · watt, Task 2 shows an improvement in both performance per watt and execution time · watt of 2.32% and 1.99%, respectively, and Task 4 shows a 3.13% improvement in execution time · watt. Since a potentially more efficient algorithm would be measured in terms of both performance and energy, we believe the improvement in the performance-energy metrics obtained by our algorithm for Tasks 1, 2, and 4, and the improvement in all metrics obtained by our algorithm for Task 1 is a positive indication. In fact, Table 4 and Figure 10(b) show that there is a positive improvement in all of the metrics tested for Task 1, all but one of the metrics tested for Task 2, and half of the metrics tested for Task 4. Also, although there is a performance loss obtained by our algorithm when executing Task 3 (tasks with eight concurrently executing benchmarks), the power savings obtained in this scenario are quite significant.

Thus our data suggest that in a majority of the cases tested, for a majority, if not all, of the measured metrics, Algorithm *CPU Scheduler* is a more efficient algorithm in terms of power savings and performance than the default Linux scheduler and CPU frequency scaling governor.

9.2.2 Evaluation of Algorithm *Priority CPU Scheduler*

In our evaluation of Algorithm 2 (*Priority CPU Scheduler*), we assumed that each process in the system has different (not necessarily unique) priorities. These priorities were static or dynamic. In other words, a process may have kept the same priority throughout its execution, or may have changed priorities during its execution cycle. These priorities were implemented using Linux nice values, and may or may not have been the default nice value of 0. Also, when surveying the effect of static priorities, we used two test trials, one in which we assigned the noncpuintensive benchmarks a lower priority, and the other in which we assigned the noncpuintensive benchmarks higher priorities. The results of this evaluation are summarized in Tables 5-17 and Figures 11-20 as follows.

For our research, we wished to investigate the performance gains or losses and power savings achieved by Algorithm *Priority CPU Scheduler* when executing a typical workload in which each process has a specific nice value (or priority), which may be static or dynamic. Thus, unlike Algorithm *CPU Scheduler*, process priority was used by this second algorithm as one of the determining factors to schedule processes. The interaction of Linux process priorities with this CPU scheduling algorithm was controlled in a manner such that process completion order, as dictated by process priority in a Linux scheduling environment, was not adversely affected. That is, both with and without using this algorithm, the completion order of processes having different priorities were the same. Both with and without using this algorithm, higher priority processes still completed before lower priority processes, despite their CPU intensity.

We assigned a nice value (priority) to each executing process. The nice values we assigned were in the range of -20 to 19, with each process's priority being inversely proportional to its nice value. When this priority was statically assigned, a process retained its nice value throughout its five execution cycles. When this priority was dynamically assigned, the first priority (Priority 1) was assigned to the process for its first three execution cycles, and its second priority (Priority 2) was assigned during its last two execution cycles. When assessing the effect of static priorities, the noncpuintensive benchmarks were assigned both high and low priorities. The priorities were unique, with the exception of those assigned to multiple instances of identical benchmarks.

As shown in Table 5, for Tasks 1-3, when assigned static priorities with the noncpuintensive processes having higher priority, the benchmarks, listed in decreasing order by priority, were the four instances of the noncpuintensive benchmark, mcf, gcc, bzip2, and sjeng. As depicted in Table 6, for the same tasks, when assigned static priorities with the noncpuintensive processes having lower priority, the benchmarks, listed in decreasing order by priority, were mcf, gcc, bzip2, sjeng, and the four instances of the noncpuintensive benchmark. Table 7 shows the first and last priorities of benchmarks in Tasks 1-3 when they were assigned dynamic priorities. The priorities for all benchmarks in Task 4 were the same as those in Tasks 1-3, with multiple instances of the same benchmark having the same nice value and priority.

We noted the completion order of each benchmark (process) in all three tasks, for both test and control groups. These results are also shown in Tables 5-8 and Figure 11. In addition to this, as shown in Table 8, due to the large number of concurrently executing processes, we gauged the completion order of benchmarks in Task 4 by placing these benchmarks in two groups, and then recording the completion order of these two groups. When assigning higher static priorities to the noncpuintensive benchmarks and when assigning dynamic priorities, Group 1 was composed of the noncpuintensive, mcf, and gcc benchmarks, and Group 2 was composed of the bzip2 and sjeng benchmarks. When assigning lower static priorities to the noncpuintensive benchmarks, Group 1 was composed of the bzip2, gcc, and mcf benchmarks, and Group 2 was composed of the noncpuintensive and sjeng benchmarks.

			Completion Order					
			Task 1		Task 2		Task 3	
Benchmark	Nice	Priority	t	c	t	c	t	c
noncpuintensive$_1$	-16	1	1	1	1	1	1	1
noncpuintensive$_2$	-16	1	–	–	–	–	2	2
noncpuintensive$_3$	-16	1	–	–	–	–	3	3
noncpuintensive$_4$	-16	1	–	–	–	–	4	4
mcf	-10	2	–	–	2	2	5	5
gcc	-4	3	2	2	3	3	6	6
bzip2	2	4	3	3	4	4	7	7
sjeng	8	5	–	–	5	5	8	8

Table 5: Completion Order of Benchmarks Evaluated (static priorities – noncpuintensive benchmarks higher priority) (t = test, c = control)

			Completion Order					
			Task 1		Task 2		Task 3	
Benchmark	Nice	Priority	t	c	t	c	t	c
mcf	-11	1	–	–	1	1	1	1
gcc	-9	2	1	1	2	2	2	2
bzip2	-7	3	2	2	3	3	3	3
sjeng	-1	4	–	–	4	4	4	4
noncpuintensive$_1$	9	5	3	3	5	5	5	5
noncpuintensive$_2$	9	5	–	–	–	–	6	6
noncpuintensive$_3$	9	5	–	–	–	–	7	7
noncpuintensive$_4$	9	5	–	–	–	–	8	8

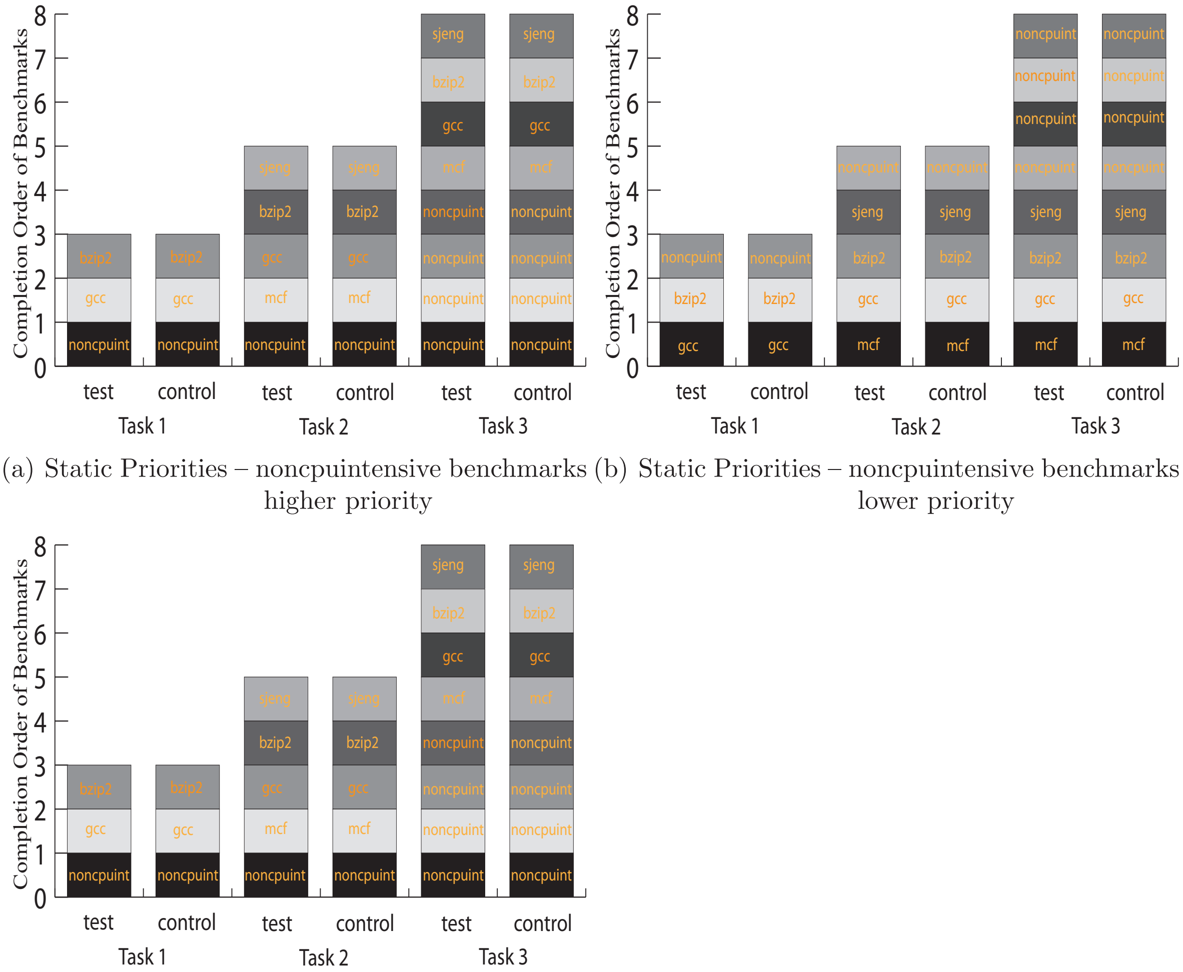
Table 6: Completion Order of Benchmarks Evaluated (static priorities – noncpuintensive benchmarks lower priority) (t = test, c = control)

					Completion Order					
	First	First	Last	Last	Task 1		Task 2		Task 3	
Benchmark	Nice	Priority	Nice	Priority	t	c	t	c	t	c
noncpuintensive$_1$	-16	1	-6	3	1	1	1	1	1	1
noncpuintensive$_2$	-16	1	-6	3	–	–	–	–	2	2
noncpuintensive$_3$	-16	1	-6	3	–	–	–	–	3	3
noncpuintensive$_4$	-16	1	-6	3	–	–	–	–	4	4
mcf	-10	2	0	5	–	–	2	2	5	5
gcc	-4	3	-14	1	2	2	3	3	6	6
bzip2	2	4	-8	2	3	3	4	4	7	7
sjeng	8	5	-2	4	–	–	5	5	8	8

Table 7: Completion Order of Benchmarks Evaluated (dynamic priorities) (t = test, c = control)

Priority Type	Group	Constituent Benchmarks	Completion Order Task 4	
			t	c
Static noncpuintensive high priority	1	noncpuintensive$_{(1-12)}$, mcf$_{(1-3)}$, gcc$_{(1-3)}$	1	1
	2	bzip2$_{(1-3)}$, sjeng$_{(1-3)}$	2	2
Static noncpuintensive low priority	1	gcc$_{(1-3)}$, bzip2$_{(1-3)}$, mcf$_{(1-3)}$	1	1
	2	noncpuintensive$_{(1-12)}$, sjeng$_{(1-3)}$	2	2
Dynamic	1	noncpuintensive$_{(1-12)}$, mcf$_{(1-3)}$, gcc$_{(1-3)}$	1	1
	2	bzip2$_{(1-3)}$, sjeng$_{(1-3)}$	2	2

Table 8: Completion Order of Benchmarks Evaluated for Task 4
(t = test, c = control)



(a) Static Priorities – noncpuintensive benchmarks higher priority

(b) Static Priorities – noncpuintensive benchmarks lower priority

(c) Dynamic Priorities

Figure 11: Completion Order of Benchmarks Evaluated

We also measured the completion times and instructions executed for each benchmark in a task, and then calculated the instructions per second (I/S) for each benchmark. These results are shown in Tables 9-11 and Figures 12-14(a)-(c). We then used the average of these values to calculate the instructions per second for the entire task. We also measured the power usage, in watts, that the system used while executing a particular task. Next we used these values to calculate two metrics, (I/S)/watts and (average process completion time) $\cdot$ watts for each task in the test and control groups. Tables 12-14 and Figures 15-17(a)-(c) depict these results. Finally, we calculated the performance gain/loss and power savings of the test versus control group, as well as the percent improvement of both metrics obtained by using Algorithm *Priority CPU Scheduler* versus the default Linux scheduler and CPU frequency scaling governor. These results are presented in Tables 15-17 and Figures 18-20(a) and (b).

As shown in Tables 5-7 and Figure 11(a)-(c), the completion order of each process in a task was the same for both the test and control groups for Tasks 1 - 3 for all three test trials. That is, for the trials where processes had static priorities in which noncpuintensive benchmarks had higher priority, static priorities in which noncpuintensive benchmarks had lower priority, and dynamic priorities, for Tasks 1 - 3, all processes completed in the same order in both the test and control groups. Also, as shown in Table 8 for Task 4, which was composed of twenty-four concurrently executing processes, for all three priority type trials, all processes in Group 1 completed execution before all processes in Group 2. This indicates that the completion order of processes is the same both using Algorithm *Priority CPU Scheduler* and without, and that this algorithm does not affect the completion order of processes that have different priorities. Both with and without our algorithm, higher priority processes still completed before lower priority processes for all four tasks, with the completion order of processes being dictated by their priorities. In both test and control groups, for Tasks 1 - 3, when the noncpuintensive benchmarks were assigned higher priorities and when all processes were assigned dynamic priorities, the completion order of processes (in increasing order) was noncpuintensive, gcc, and bzip2 for Task 1, noncpuintensive, mcf, gcc, bzip2, and sjeng for Task 2, and all four instances of noncpuintensive, mcf, gcc, bzip2, and sjeng for Task 3. When the noncpuintensive benchmarks were assigned lower priorities, in both test and control groups for Tasks 1 - 3, the completion order of processes (in increasing order) was gcc, bzip2, and noncpuintensive for Task 1, mcf, gcc, bzip2, sjeng, and noncpuintensive for Task 2, and mcf, gcc, bzip2, sjeng, and all four instances of noncpuintensive for Task 3. As mentioned, for Task 4 with all three priority types, all Group 1 processes completed execution before all Group 2 processes. This indicates that Algorithm *Priority CPU Scheduler* does not adversely affect the Linux priority scheduling of processes as dictated by their nice values.

Thus our algorithm interacts with a process's priority in a manner that does not negatively affect its completion order. This is due to the fact that in our algorithm, a higher priority process will be context switched to a cpuset with faster frequency cores, if it is CPU intensive and its nice value is less than 9, (and also if it is non-CPU intensive, even if the load average is almost as high as *num_cpus*). Also, a lower priority process will not be context switched to higher frequency cores unless it is non-CPU intensive, its nice value is greater than -11, and the load average is less than 0.15 * *num_cpus* (and also if it is CPU intensive and its nice value is greater than or equal to 9).

Instructions/Second (in billions)

	Task 1		Task 2		Task 3	
	test	control	test	control	test	control
bzip2	2.62	2.63	1.77	2.43	2.32	2.32
gcc	1.06	1.23	1.24	1.10	1.01	0.96
mcf	–	–	0.33	0.26	0.21	0.33
sjeng	–	–	1.38	2.34	1.38	2.15
noncpuintensive$_1$	1.01	0.60	0.56	0.68	0.96	0.91
noncpuintensive$_2$	–	–	–	–	0.97	0.88
noncpuintensive$_3$	–	–	–	–	1.13	0.88
noncpuintensive$_4$	–	–	–	–	1.12	0.90

Table 9: Performance of Benchmarks Evaluated (static priorities – noncpuintensive benchmarks higher priority) for Algorithm 2

Instructions/Second (in billions)

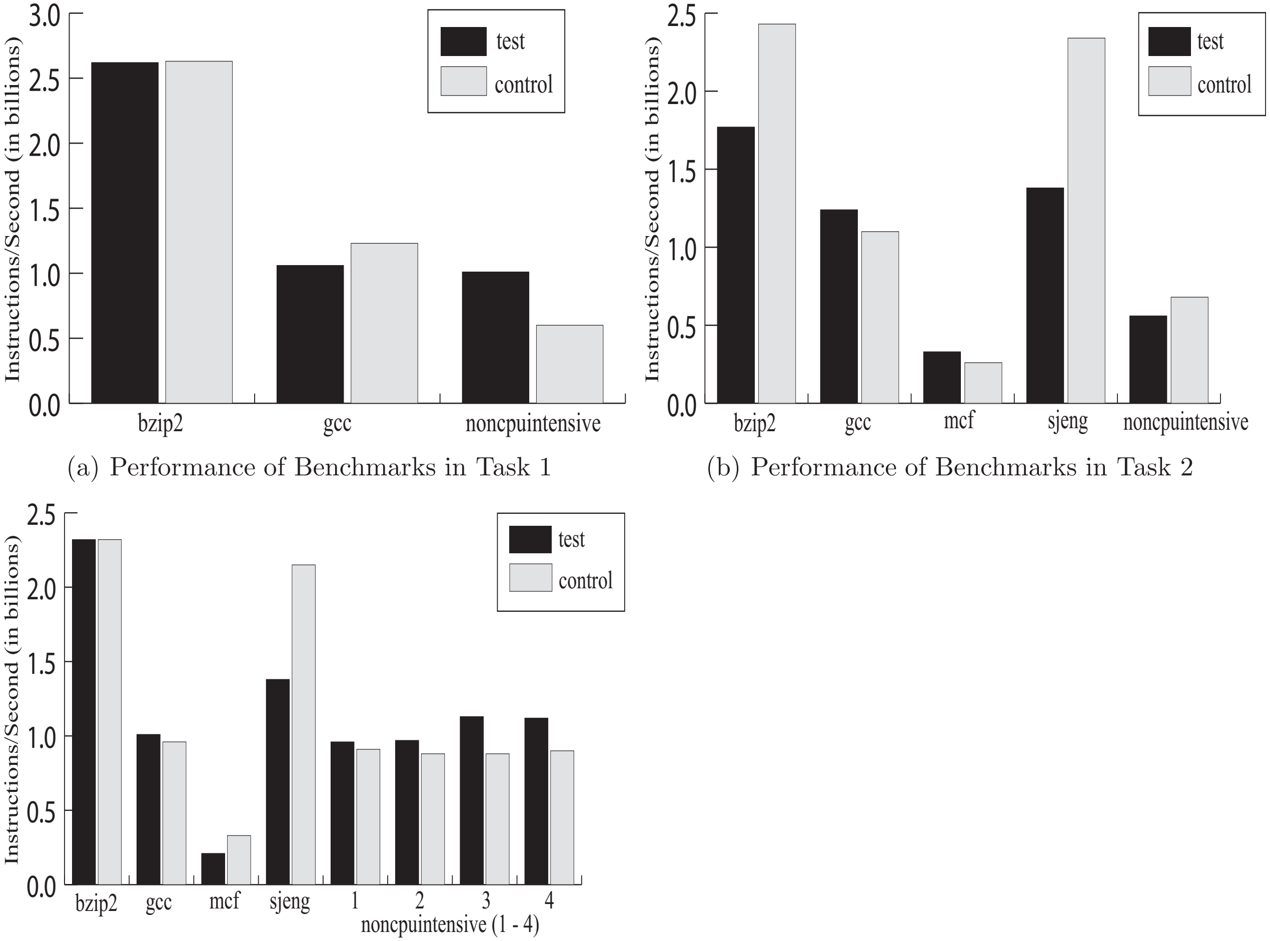
	Task 1		Task 2		Task 3	
	test	control	test	control	test	control
bzip2	1.61	2.56	2.51	2.43	1.83	2.22
gcc	1.01	1.17	1.21	1.10	0.97	0.96
mcf	–	–	0.18	0.26	0.24	0.33
sjeng	–	–	1.75	2.34	1.38	2.27
noncpuintensive$_1$	1.57	0.62	1.14	0.68	0.69	0.69
noncpuintensive$_2$	–	–	–	–	0.69	0.68
noncpuintensive$_3$	–	–	–	–	0.69	0.68
noncpuintensive$_4$	–	–	–	–	0.69	0.69

Table 10: Performance of Benchmarks Evaluated (static priorities – noncpuintensive benchmarks lower priority) for Algorithm 2

Instructions/Second (in billions)

	Task 1		Task 2		Task 3	
	test	control	test	control	test	control
bzip2	2.62	2.57	2.00	2.48	2.18	2.31
gcc	1.06	1.23	1.26	1.10	1.04	1.17
mcf	–	–	0.18	0.16	0.21	0.30
sjeng	–	–	1.74	2.24	1.35	2.13
noncpuintensive$_1$	1.04	0.60	0.65	0.61	0.40	0.85
noncpuintensive$_2$	–	–	–	–	0.73	0.84
noncpuintensive$_3$	–	–	–	–	0.60	0.85
noncpuintensive$_4$	–	–	–	–	0.60	0.86

Table 11: Performance of Benchmarks Evaluated (dynamic priorities) for Algorithm 2

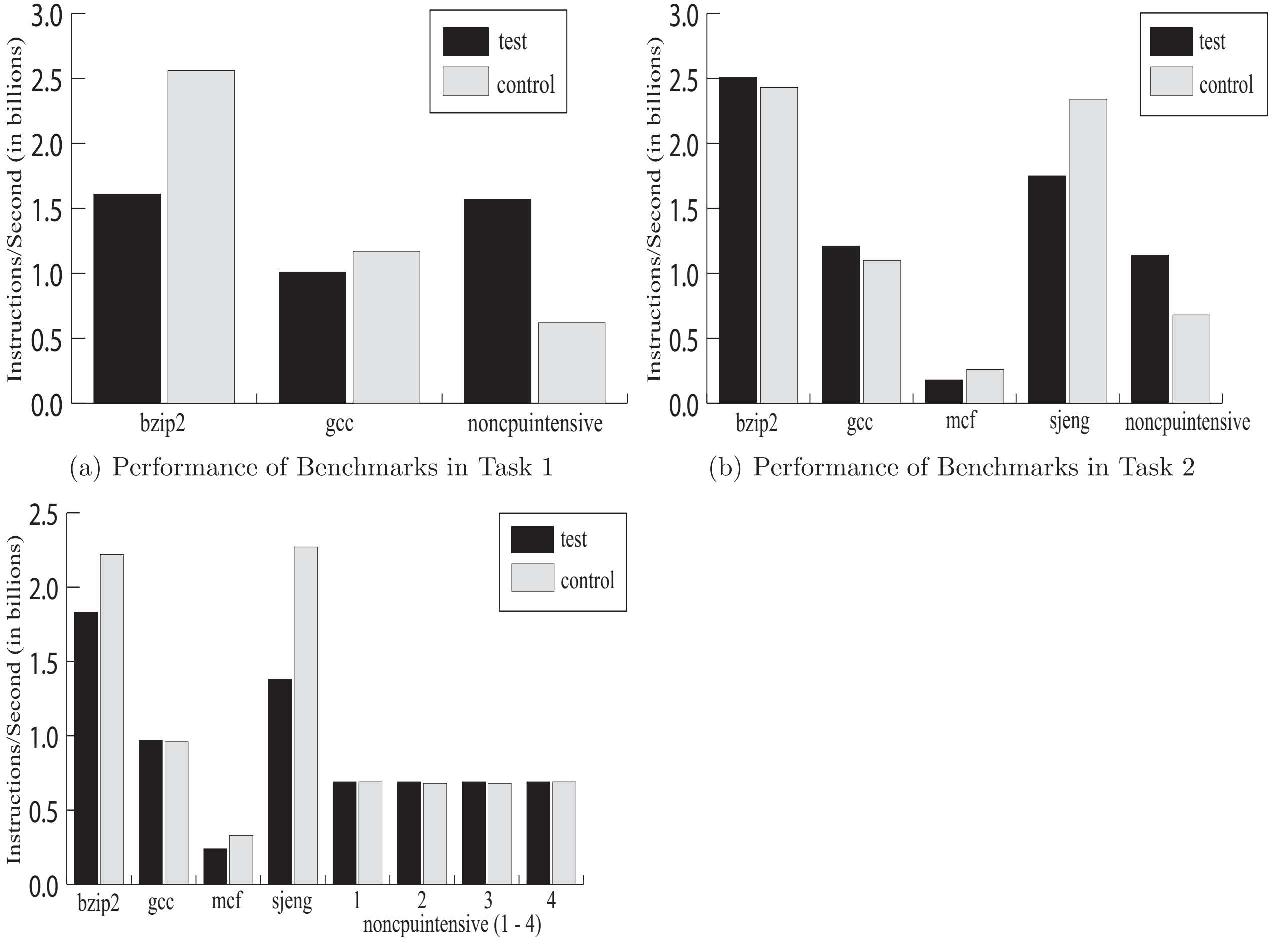


(a) Performance of Benchmarks in Task 1

(b) Performance of Benchmarks in Task 2

(c) Performance of Benchmarks in Task 3

Figure 12: Performance of Benchmarks Evaluated (static priorities – noncpuintensive benchmarks higher priority) for Algorithm 2

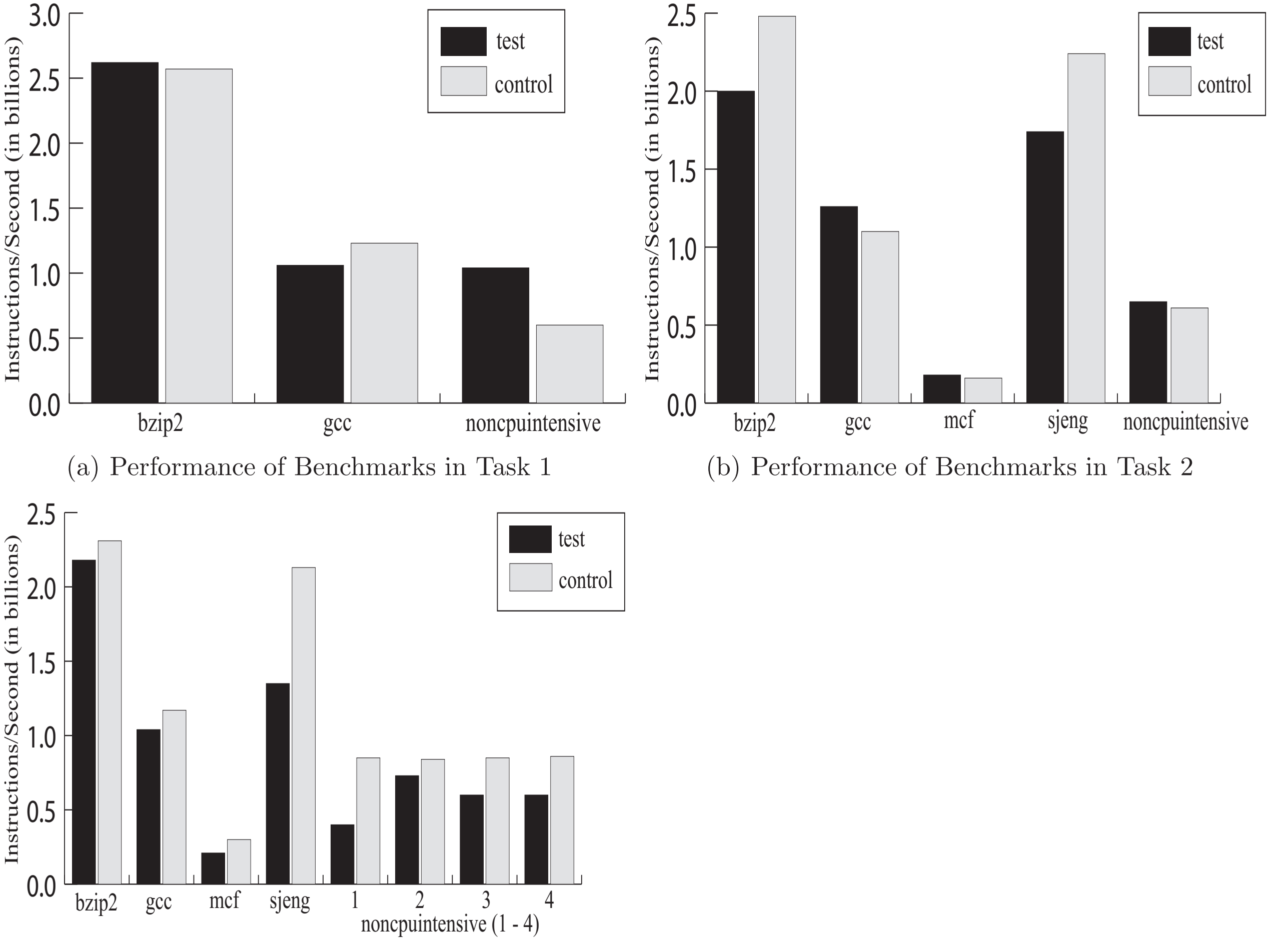


(a) Performance of Benchmarks in Task 1

(b) Performance of Benchmarks in Task 2

(c) Performance of Benchmarks in Task 3

Figure 13: Performance of Benchmarks Evaluated (static priorities – noncpuintensive benchmarks lower priority) for Algorithm 2



(a) Performance of Benchmarks in Task 1

(b) Performance of Benchmarks in Task 2

(c) Performance of Benchmarks in Task 3

Figure 14: Performance of Benchmarks Evaluated (dynamic priorities) for Algorithm 2

Tables 9-11 and Figures 12-14 indicate that the higher CPU intensive processes bzip2, sjeng, and, to a lesser degree, gcc, have a higher instructions per second value than mcf and noncpuintensive, which are less CPU intensive and more memory intensive, for all three priority type trials. We believe this is caused by two factors. Firstly, memory performance saturation did occur when executing memory intensive benchmarks, especially mcf. Thus the delay created by waiting for data from memory caused a decrease in the overall instructions per second rate for those processes. Secondly, the higher CPU intensive processes were being context switched to cpusets with higher frequency cores by Algorithm *Priority CPU Scheduler*. Our algorithm context switches CPU intensive processes (whose performance index is true and whose nice value is less than 9) to the higher frequency cpusets and, in a heavily loaded system, moves less CPU intensive processes to lower frequency cores. Since all three of these benchmarks had a nice value less than 9, they would have been context switched to higher frequency cores if their performance index was true. Also, if a higher frequency cpuset has cores with an empty ready queue, our algorithm gives priority to higher CPU intensive, higher priority processes to be assigned to that cpuset. These two characteristics allow Algorithm *Priority CPU Scheduler* to advantageously schedule processes with the goal of lowering the execution time and improving the performance of the entire task.

Also, as shown in Tables 10 and 11 and Figures 13 and 14 (a) and (b), for Task 1 and Task 2, when the noncpuintensive benchmarks had a lower priority, the noncpuintensive benchmark within the test group had a higher instructions per second value than the control group. We believe this performance gain by the noncpuintensive benchmark was obtained for three reasons. Firstly, if a noncpuintensive process is the only process in the system or in a lightly loaded system, the load average will be less than 0.15 * num_cpus. In this scenario, our algorithm will context switch this non-CPU intensive process, despite its lower priority, to higher frequency cores, enabling its faster execution, whereas an "on demand" governor will lower the CPU frequency of the cores upon which it is executing, thereby lowering its execution speed. Secondly, in our algorithm, if two or more cpusets have empty ready queues, and if this non-CPU intensive process is the only process in the system, it will be context switched to the cpuset with CPU's operating at the highest frequency, thereby speeding up its execution and improving performance. We did not see this same occurrence in Task 3 because the system was heavily loaded, since there were eight concurrently executing benchmarks, and the lower priority non-CPU intensive benchmarks were not context switched to faster frequency cores because the load average was greater than 0.15 * *num_cpus*. The third reason is that our algorithm allows the load average to be as high as *num_cpus* before a non-CPU intensive process having a nice value > -1, is context switched to lower frequency cores. This allows non-CPU intensive processes to execute upon cores operating at a higher frequency than they would if they were executing upon a system with an "on demand" frequency scaling governor, thus enabling a performance gain.

As shown in Table 9 and Figure 12, when noncpuintensive processes were assigned higher priorities, for Task 3, the noncpuintensive benchmarks in the test group exhibited a higher instructions per second value than their control counterparts. We believe this is true because our algorithm allows the load average to be as high as *num_cpus* if a non-CPU intensive process's priority is less than or equal to -11, before it context switches this process to higher frequency cores. Since a system with eight cores and eight concurrently executing benchmarks does have a load average equal to *num_cpus*, our algorithm will context switch such a process to higher frequency cores.

	Task 1		Task 2		Task 3		Task 4	
	test	control	test	control	test	control	test	control
Performance Avg Ins/Sec (in billions)	1.56	1.49	1.06	1.38	1.14	1.17	0.67	1.00
Performance per Watt Avg (Ins/Sec)/Watt (in millions)	12.21	10.79	7.54	8.76	7.90	7.24	4.63	5.44
Execution Time · Watt Avg Time (Sec) · Watt (in thousands)	16.17	17.35	24.22	20.85	16.85	16.30	34.04	28.84

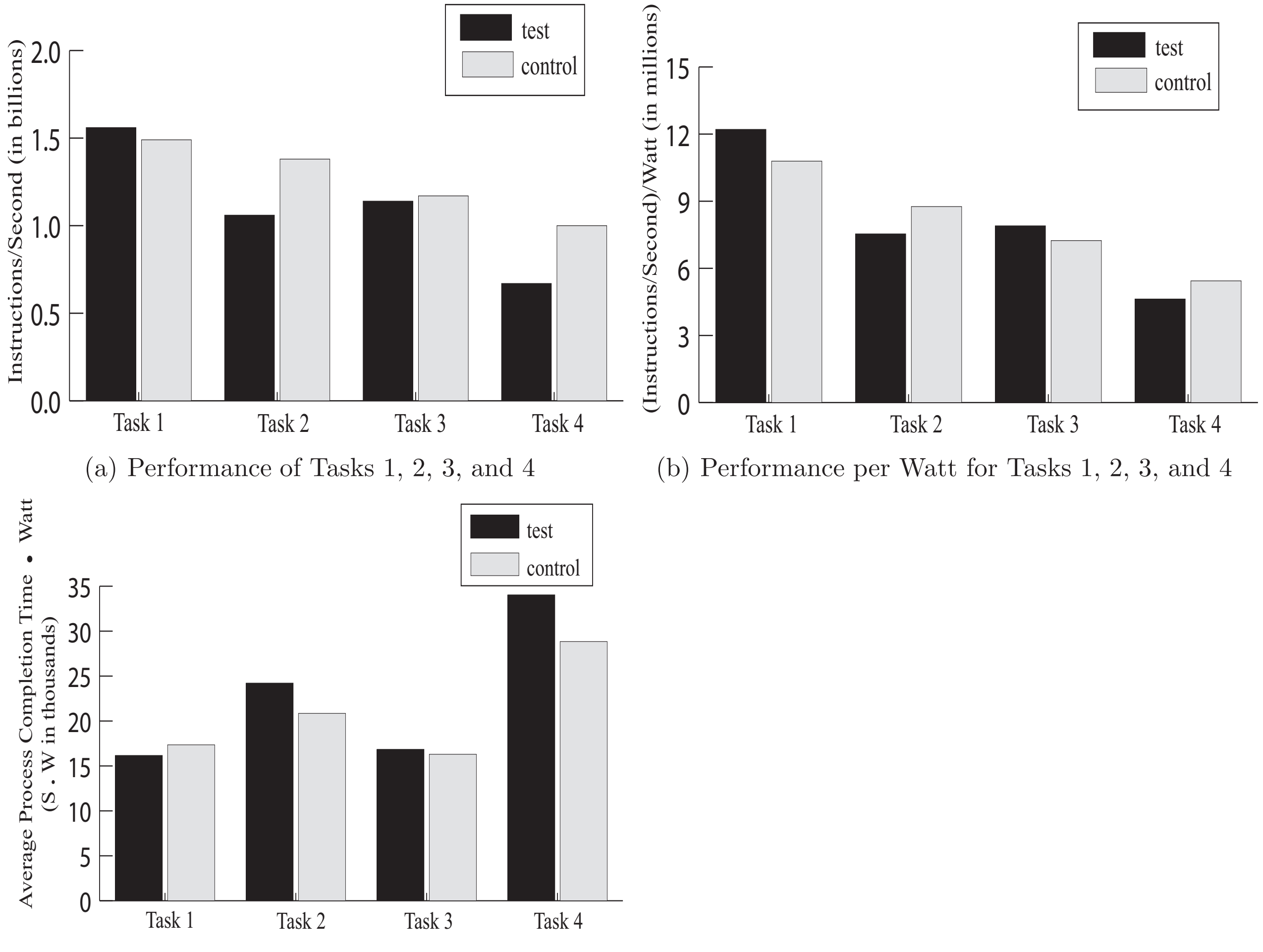
Table 12: Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (static priorities – noncpuintensive benchmarks higher priority)

	Task 1		Task 2		Task 3		Task 4	
	test	control	test	control	test	control	test	control
Performance Avg Ins/Sec (in billions)	1.40	1.45	1.36	1.36	0.90	1.07	0.47	0.71
Performance per Watt Avg (Ins/Sec)/Watt (in millions)	10.66	10.36	9.70	8.41	6.11	5.83	3.22	3.79
Execution Time · Watt Avg Time (Sec) · Watt (in thousands)	38.40	67.72	36.24	53.29	83.66	105.75	103.00	86.65

Table 13: Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (static priorities – noncpuintensive benchmarks lower priority)

	Task 1		Task 2		Task 3		Task 4	
	test	control	test	control	test	control	test	control
Performance Avg Ins/Sec (in billions)	1.57	1.47	1.16	1.36	0.89	1.16	0.63	0.67
Performance per Watt Avg (Ins/Sec)/Watt (in millions)	12.01	10.26	8.06	8.57	6.08	7.01	4.33	3.66
Execution Time · Watt Avg Time (Sec) · Watt (in thousands)	16.58	18.19	23.67	21.29	19.14	16.38	36.39	38.26

Table 14: Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (dynamic priorities)

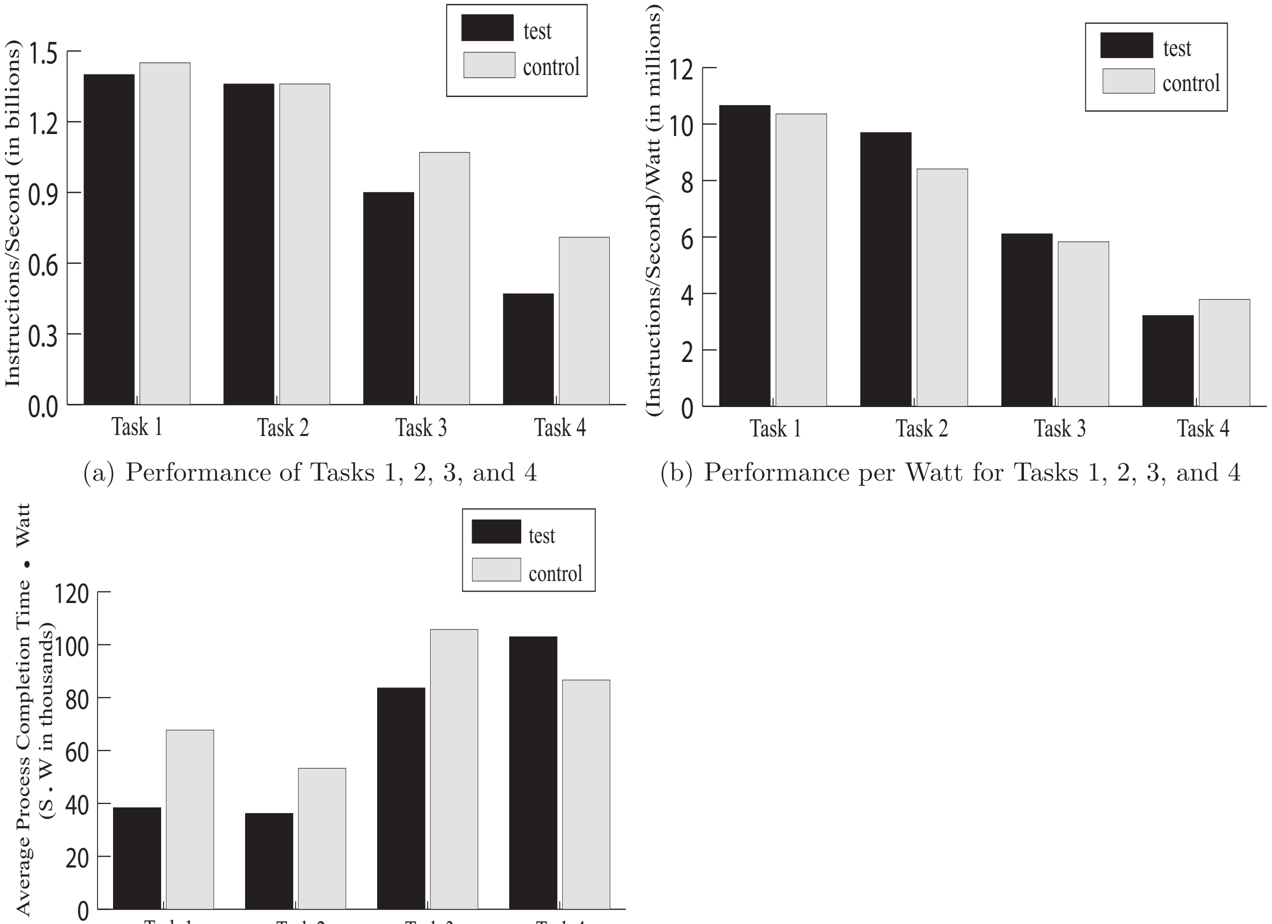


(a) Performance of Tasks 1, 2, 3, and 4

(b) Performance per Watt for Tasks 1, 2, 3, and 4

(c) Execution Time · Watt for Tasks 1, 2, 3, and 4

Figure 15: Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (static priorities – noncpuintensive benchmarks higher priority)

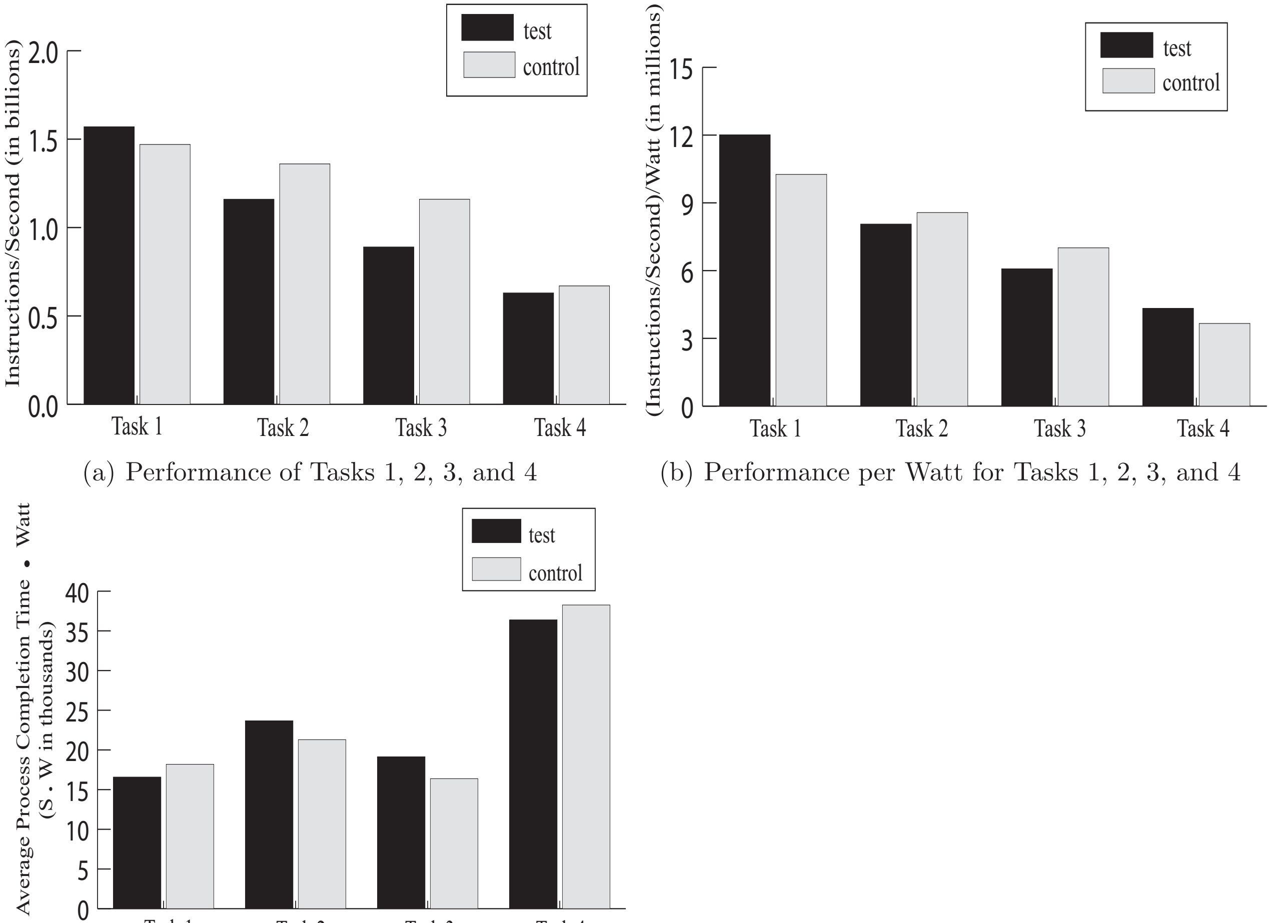


(a) Performance of Tasks 1, 2, 3, and 4

(b) Performance per Watt for Tasks 1, 2, 3, and 4

(c) Execution Time · Watt for Tasks 1, 2, 3, and 4

Figure 16: Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (static priorities – noncpuintensive benchmarks lower priority)



(a) Performance of Tasks 1, 2, 3, and 4

(b) Performance per Watt for Tasks 1, 2, 3, and 4

(c) Execution Time · Watt for Tasks 1, 2, 3, and 4

Figure 17: Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, and 3 (dynamic priorities)

In addition to this, Figures 15 - 17(a) show that the instructions per second values for the test groups for Task 1 were higher than the control groups for the trials where the noncpuintensive benchmarks had higher priority and where benchmarks had dynamic priorities, and were almost the same for the trial where the noncpuintensive benchmarks had lower priority. Also, for all other tasks and priority type trials, the test groups' values were not much lower than the control groups. In only one test case (Task 4, noncpuintensive benchmarks with higher priorities) was this value no less than 0.33 of the control group's value. This may be due to the fact that our algorithm allows processes to execute upon CPU cores that are better suited to their current execution characteristics. Therefore, our algorithm can produce an instructions per second rate that is comparable to that produced by the default Linux scheduler and "on demand" governor, despite any context switching overhead.

Also, as shown in Tables 12-14 and Figures 15-17(b), the metric performance per watt for the majority of the tasks in all three priority type trials was higher for the test group than the control group. In addition, as indicated by Figures 15-17(c), the metric execution time · watt was lower for half of the tasks in all three priority type trials for the test group than the control group. Also, for Tasks 1, 2, and 3 in which benchmarks had static priorities with noncpuintensive benchmarks having lower priorities, the test group had a lower value for the metric execution time · watt than the control group. Since a higher value for the first metric and a lower value for the second metric indicate a potentially more efficient algorithm, when taking into account both power savings and performance, we believe Algorithm *Priority CPU Scheduler* is more efficient than the default Linux scheduler and "on demand" CPU frequency scaling governor for a majority of the cases tested.

	Task 1	Task 2	Task 3	Task 4
Total Power Savings (Watts)	10	17	17	39
Power Savings (%)	7.25	10.83	10.56	21.31
Performance Improvement (%)	4.76	-23.21	-2.46	-33.04
Performance per Watt Improvement (%)	11.66	-13.89	8.30	-14.90
Execution Time · Watt Improvement (%)	6.77	-13.90	-3.24	-15.28

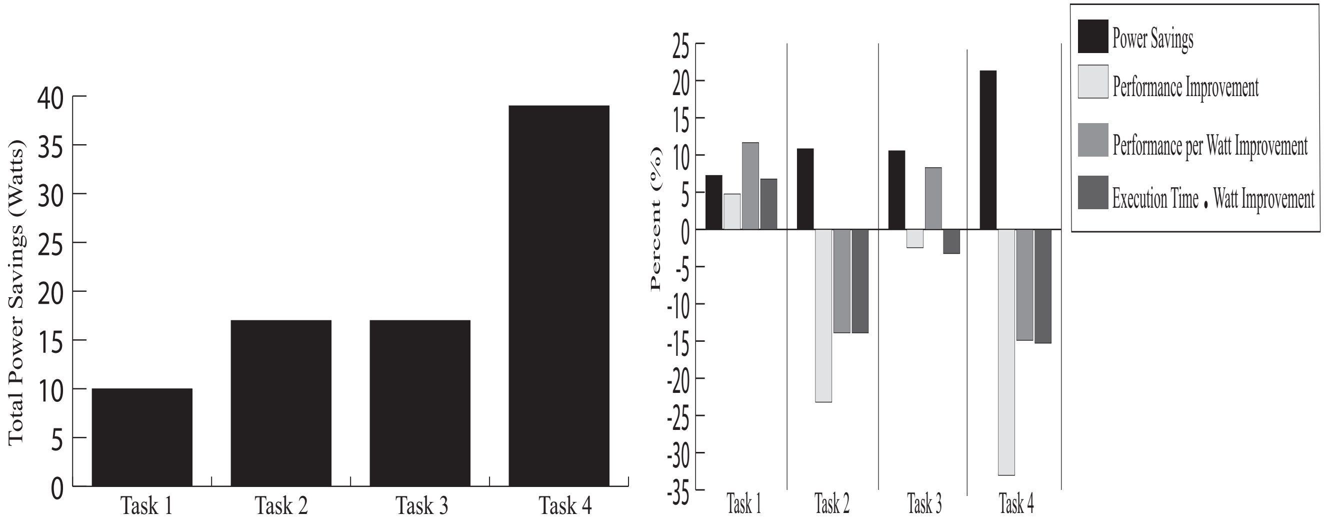
Table 15: Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (static priorities – noncpuintensive benchmarks higher priority)

	Task 1	Task 2	Task 3	Task 4
Total Power Savings (Watts)	9	22	36	41
Power Savings (%)	6.43	13.58	19.67	21.93
Performance Improvement (%)	-3.75	-0.40	-15.75	-33.6
Performance per Watt Improvement (%)	2.78	13.23	4.65	-14.95
Execution Time · Watt Improvement (%)	43.29	32.00	20.89	-15.88

Table 16: Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (static priorities – noncpuintensive benchmarks lower priority)

	Task 1	Task 2	Task 3	Task 4
Total Power Savings (Watts)	12	15	20	37
Power Savings (%)	8.39	9.43	12.05	20.33
Performance Improvement (%)	6.73	-14.8	-23.67	-5.55
Performance per Watt Improvement (%)	14.56	-5.93	-13.21	15.64
Execution Time · Watt Improvement (%)	8.84	-10.06	-14.42	4.89

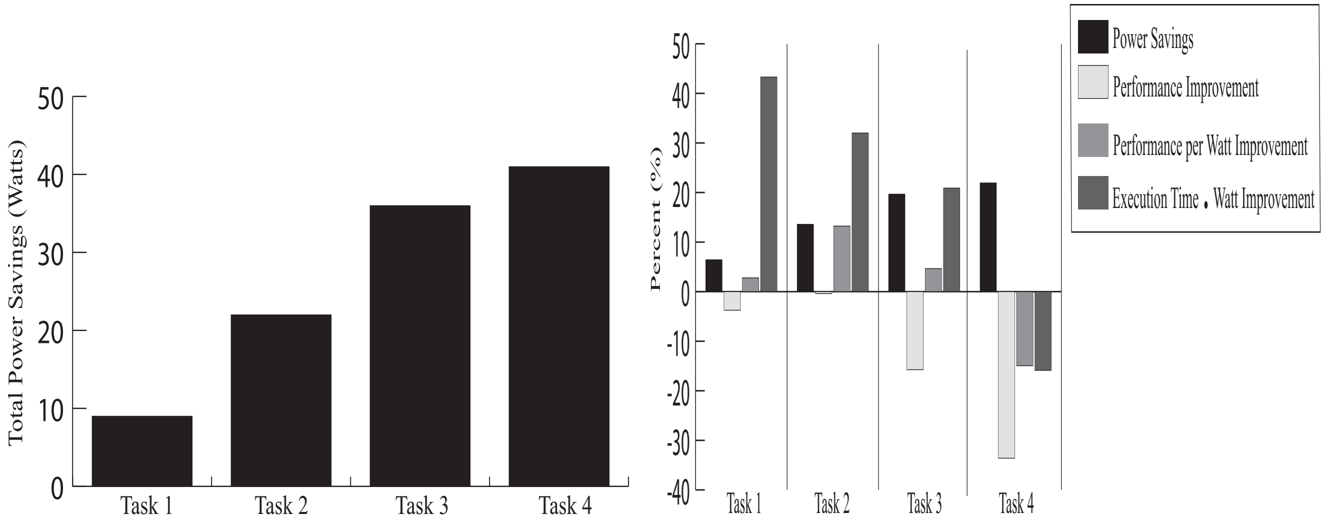
Table 17: Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, and 3 (dynamic priorities)



(a) Total Power Savings for Tasks 1, 2, 3, and 4

(b) Percent Power Savings and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4

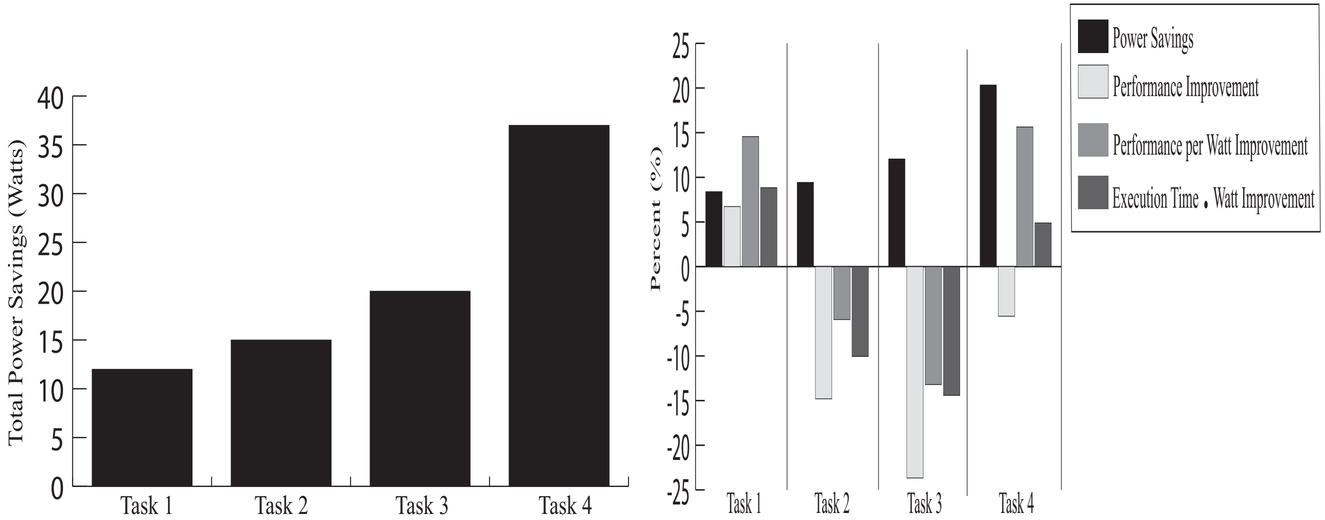
Figure 18: Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (static priorities – noncpuintensive benchmarks higher priority)



(a) Total Power Savings for Tasks 1, 2, 3, and 4

(b) Percent Power Savings and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4

Figure 19: Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4 (static priorities – noncpuintensive benchmarks lower priority)



(a) Total Power Savings for Tasks 1, 2, 3, and 4

(b) Percent Power Savings and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, 3, and 4

Figure 20: Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt for Tasks 1, 2, and 3 (dynamic priorities)

As indicated by Tables 15-17 and Figures 18-20(a), the total power savings for Tasks 1, 2, 3, and 4 obtained by using Algorithm *Priority CPU Scheduler* was 10, 17, 17, and 39 watts when processes were assigned static priorities with noncpuintensive benchmarks having higher priorities, 9, 22, 36, and 41 watts when processes had static priorities with noncpuintensive benchmarks having lower priorities, and 12, 15, 20, and 37 watts when processes had dynamic priorities. This shows that the power savings obtained by our algorithm is significant and becomes even more significant as the number of concurrently executing processes is greater. The percent power savings, relative to the control group, is also significant, as shown in Tables 15-17 and Figures 18-20(b). It can be as high as 21.93%, or 41 watts. For all four tasks, there is a significant increase in power savings obtained by our algorithm.

As shown by Tables 15-17 and Figures 18-20(b), while there is a performance loss associated with using Algorithm *Priority CPU Scheduler* when considering only instructions per second, for some cases, there is an improvement in performance for Task 1 when processes were assigned static priorities (noncpuintensive benchmarks having higher priority) and processes were assigned dynamic priorities. Also, Table 16 and Figure 19(b) show that there is a positive improvement in the performance per watt and execution time · watt for Tasks 1 - 3 when processes were assigned static priorities (noncpuintensive benchmarks having lower priority). In fact, there is a very significant improvement in the performance per watt for Task 2 (13.23%) and the execution time · watt for Tasks 1 - 3 (43.29%, 32%, and 20.89%), in this priority type trial. Also, there is a positive improvement in the performance per watt and execution time · watt for the majority of the test cases for all four tasks and all three priority type trials. Since a potentially more efficient algorithm would be measured in terms of both performance and energy, we believe the improvement in the performance-energy metrics, obtained by our algorithm, for these scenarios is a positive indication that outweighs the loss of performance observed when taking into account just instructions per second. Also, although there is a performance loss (when only considering instructions per second) obtained by our algorithm, the power savings obtained are quite significant.

Thus our data suggest that in all of the cases tested, for a majority of the measured metrics, Algorithm *Priority CPU Scheduler* is a more efficient algorithm in terms of power savings and performance than the default Linux scheduler and CPU frequency scaling governor. Also, the interaction of our algorithm with nice (priority) values is such that the completion order of processes, as dictated by their nice values, is not affected for all four tasks, in both a lightly-loaded and heavily-loaded system (one with twenty-four concurrently executing benchmarks). This indicates that our algorithm does not adversely impact process completion order when interacting with process priority. Higher priority processes still complete before lower priority processes, both with and without our algorithm, in both a lightly loaded and heavily loaded system. However, in this scenario, Algorithm *Priority CPU Scheduler* ensures both significant power savings and, in a majority of the cases, an increase in the performance-energy metrics.

9.2.3 Evaluation of Algorithm *Cache Miss Priority CPU Scheduler*

In our evaluation of Algorithm 3 (*Cache Miss Priority CPU Scheduler*), we again assumed that each process in the system has different (not necessarily unique) priorities. To account for the possibility that a process may change priorities during its execution cycle, we assigned dynamic priorities to these processes. These priorities were implemented using Linux nice values, and may or may not have been the default nice value of 0. Also, in our survey, we measured the number of cache misses, cache references, and cache miss to cache reference ratio generated by each executing process. We measured these values for a task containing twenty-four concurrently executing processes (the same benchmarks as Task 4 in section 9.2.2), and then calculated the average number of cache misses, cache references, and average cache miss to cache reference ratio for the entire task. We then compared these values to those for the control case and to Algorithm 2, when using Task 4 with the dynamic priority test trial. The results of this evaluation are summarized in Tables 18-21 and Figures 21-23 as follows.

Unlike Algorithm *Priority CPU Scheduler* , Algorithm *Cache Miss Priority CPU Scheduler* uses the ratio between the number of cache misses to the number of cache references generated by a process at runtime as one of the determining factors to schedule processes. If a process is context switched too often and consequently changes the CPU upon which it executes often, it will have a tendency to generate a larger number of cache misses, since it must access a different CPU's cache after it is context switched. To prevent this, Algorithm *Cache Miss Priority CPU Scheduler* does not context switch such a process to cores operating at a higher frequency if it generates too many cache misses. Thus, if the number of cache misses is too high, then a different strategy is implemented by Algorithm *Cache Miss Priority CPU Scheduler*. This new strategy's goal is to improve performance by reducing the overhead associated with cache access, while still maintaining power savings.

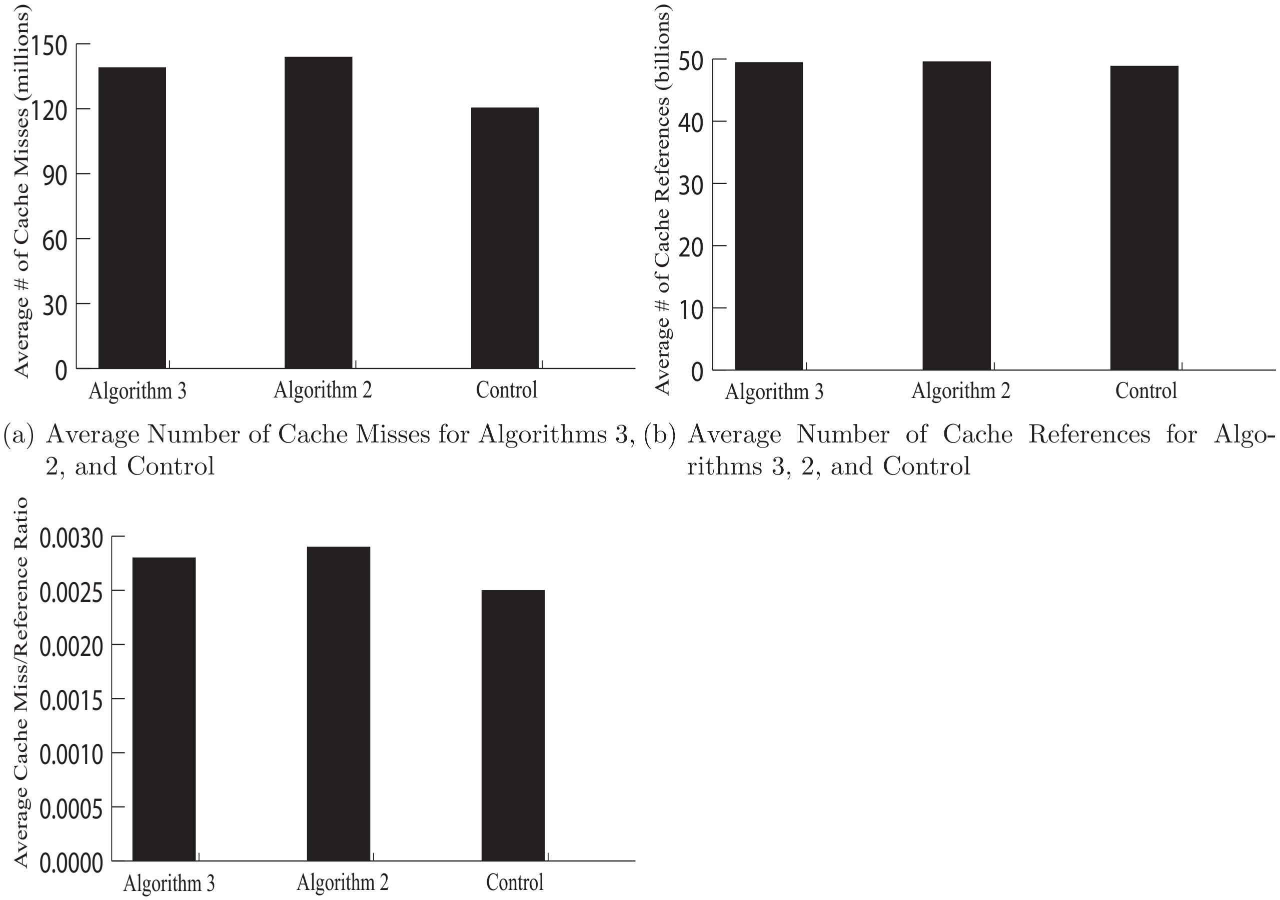
In our test case, we assigned dynamic priorities to our processes, and in order to test the scenario where Algorithms 1 and 2 exhibited the worst performance (which was Task 4), we used twenty-four concurrently executing processes. Like the dynamic priorities trial type for the first two algorithms, the first priority (Priority 1) was assigned to a process for its first three execution cycles, and its second priority (Priority 2) was assigned during its last two execution cycles. Also, these priorities were unique, with the exception of those assigned to multiple instances of identical benchmarks. Since these priorities were the same as those used for Algorithm 2, they are shown in Table 7. Again, like the control group used for Algorithm 2, our control group consisted of the same processes executing with the default Linux scheduler and CPU frequency scaling governor.

We calculated the average number of cache misses, cache references, and average cache miss to cache reference ratio for our task obtained by Algorithm *Cache Miss Priority CPU Scheduler*, Algorithm *Priority CPU Scheduler*, and by the control group. These results are shown in Table 18 and Figure 21.

We also measured the instructions per second (I/S) for each benchmark and used the average of these values to calculate the instructions per second for the entire task. In addition, we measured the power usage, in watts, that the system used while executing our task. We then used these values to calculate two metrics, (I/S)/watts and (Average process completion time) · watts for our task in the test and control groups. Table 19 and Figure 22(a)-(c) depict these results. Also, we calculated the performance gain/loss and power savings of the test versus control group, as well as the percent improvement of both performance metrics obtained by using Algorithm *Cache Miss Priority CPU Scheduler* versus the default Linux scheduler and CPU frequency scaling governor. These results are presented in Table 20 and Figure 22(d). Finally, we calculated the percent reduction of the average number of cache misses, cache references, and average cache miss to cache reference ratio obtained by Algorithms 3 and 2 when compared to the default Linux scheduler and CPU frequency scaling governor (control group) and that obtained by Algorithm *Cache Miss Priority CPU Scheduler* when compared to Algorithm *Priority CPU Scheduler*. These results are shown in Table 21 and Figure 23.

	Algorithm 3	Algorithm 2	Control
Average Number of Cache Misses (in millions)	139.00	143.83	120.40
Average Number of Cache References (in billions)	49.45	49.58	48.86
Average Cache Miss/Reference Ratio	.0028	.0029	.0025

Table 18: Average Number of Cache Misses, Cache References, and Average Cache Miss/Reference Ratio for Algorithms 3, 2, and Control



(a) Average Number of Cache Misses for Algorithms 3, 2, and Control

(b) Average Number of Cache References for Algorithms 3, 2, and Control

(c) Average Cache Miss/Reference Ratio for Algorithms 3, 2, and Control

Figure 21: Average Number of Cache Misses, Cache References, and Cache Miss/Reference Ratio for Algorithms 3, 2, and Control for Task 4 (dynamic priorities)

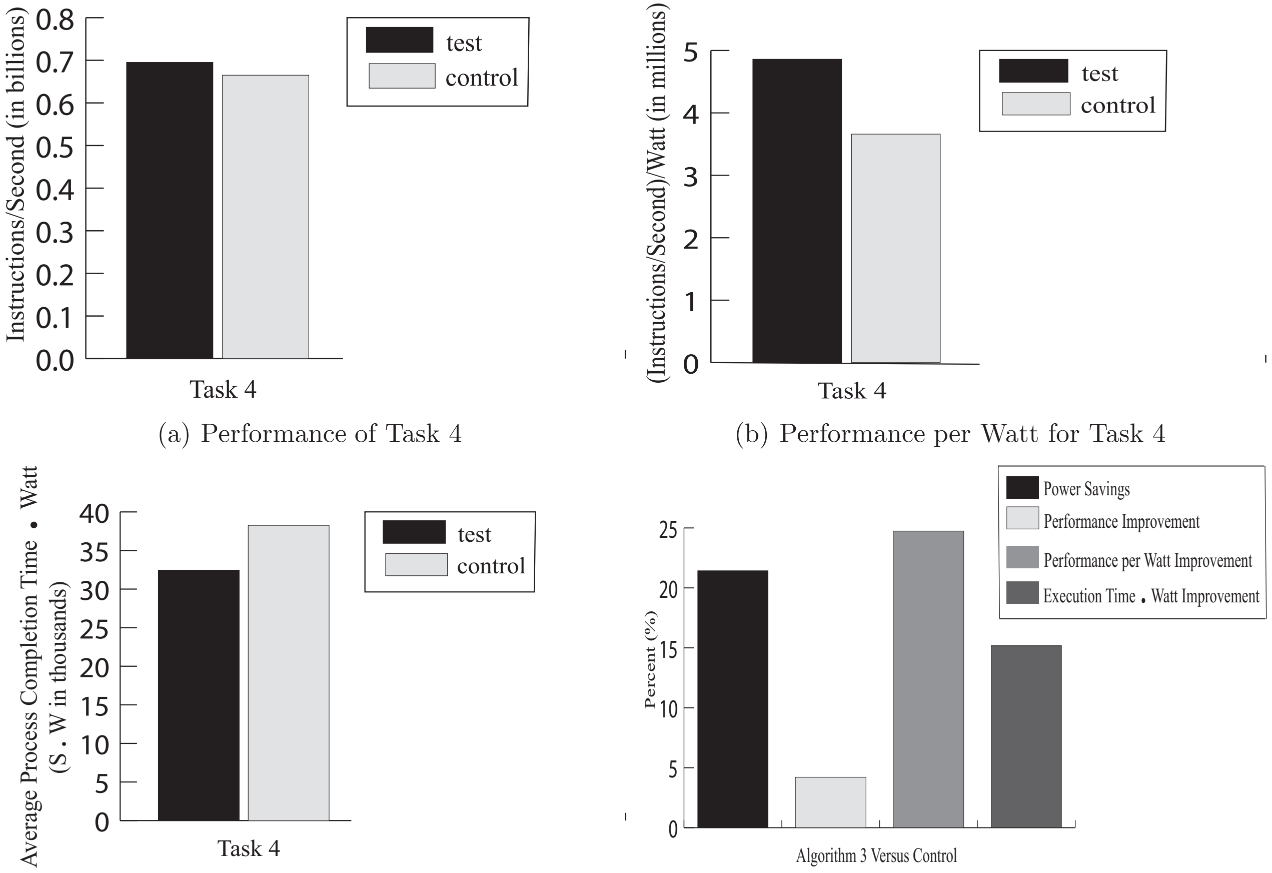
Table 18 and Figure 21 show that the average number of cache misses, average number of cache references, and average cache miss/reference ratio was lower for Algorithm *Cache Miss Priority CPU Scheduler* when compared to that obtained by Algorithm *Priority CPU Scheduler*. This indicates that there is a definite reduction in the number of cache misses and cache miss to cache reference ratio obtained by Algorithm *Cache Miss Priority CPU Scheduler*. This reduction is achieved because this algorithm does not context switch CPU intensive processes to higher frequency cores if their nice values are less than 9, their performance indices are true, and if their cache miss indices are true, or if non-CPU intensive processes have nice values less than or equal to -11, if the load average is less than or equal to *num_cpus*, if their performance indices are true, and if their cache miss indices are true, and if they have nice values greater than -11, the load average is less than 0.15 * *num_cpus*, their performance indices are true, and if their cache miss indices are true. Since a process's cache miss index represents its cache miss to cache reference ratio, Algorithm *Cache Miss Priority CPU Scheduler* will not context switch processes that generate many cache misses, thus reducing context switching overhead and improving performance.

	Task 4	
	test	control
Performance Avg Ins/Sec (in billions)	0.69	0.67
Performance per Watt Avg (Ins/Sec)/Watt (in millions)	4.86	3.66
Execution Time · Watt Avg Time (Sec) · Watt (in thousands)	32.45	38.26

Table 19: Performance, Performance per Watt, and Execution Time · Watt for Task 4 (dynamic priorities)

	Task 4
Total Power Savings (Watts)	39
Power Savings (%)	21.43
Performance Improvement (%)	4.21
Performance per Watt Improvement (%)	24.74
Execution Time · Watt Improvement (%)	15.18

Table 20: Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt by Algorithm 3 for Task 4 (dynamic priorities)



(a) Performance of Task 4

(b) Performance per Watt for Task 4

(c) Execution Time · Watt for Task 4

(d) Percent Power Savings and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt by Algorithm 3 (Task 4)

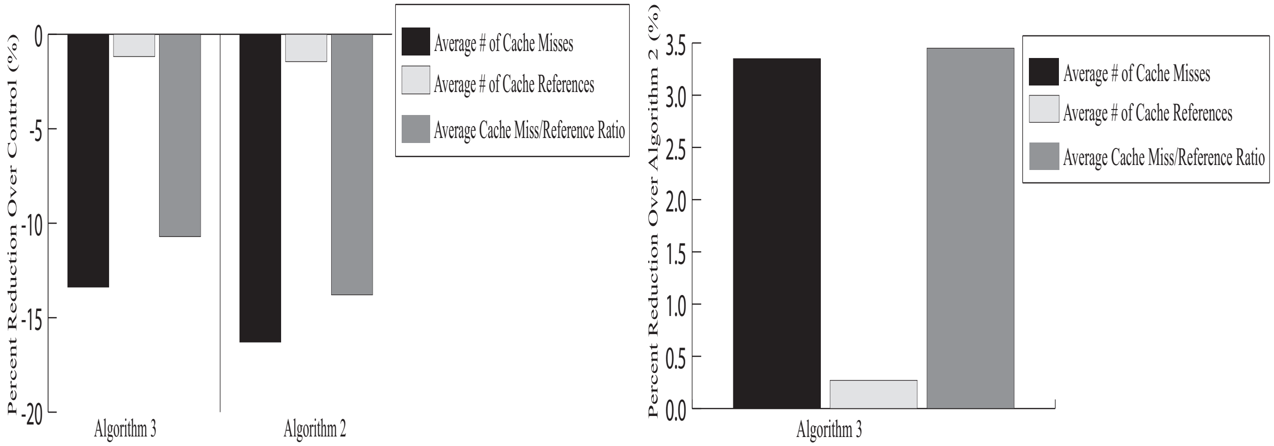
Figure 22: Performance, Performance per Watt, Execution Time · Watt, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt by Algorithm 3 for Task 4 (dynamic priorities)

As shown in Table 19 and Figure 22(a)-(c), performance, in terms of average instructions per second, and performance per watt were both higher for the test group than the control group by Algorithm *Cache Miss Priority CPU Scheduler* for Task 4. Also, the metric execution time · watt was lower for the test group than the control group. Since a higher value for the first metric and a lower value for the latter imply a more efficient algorithm, we believe Algorithm *Cache Miss Priority CPU Scheduler* is a more efficient algorithm in terms of both power savings and performance than the default Linux CPU frequency scaling governor.

Also, Table 20 and Figure 22(d) show that there is a significant increase in the percent power savings and percent improvement of performance, performance per watt, and execution time · watt for Algorithm *Cache Miss Priority CPU Scheduler* when compared to control. Also, there is a very significant total power savings of 39 watts. This indicates that the strategy employed by Algorithm *Cache Miss Priority CPU Scheduler* is an effective means to both increase power savings and improve performance significantly.

	Percent Reduction Over Control		Percent Reduction Over Algorithm 2
	by Algorithm 3	by Algorithm 2	by Algorithm 3
Average Number of Cache Misses (%)	-13.38	-16.29	3.35
Average Number of Cache References (%)	-1.18	-1.45	0.27
Average Cache Miss to Cache Reference Ratio (%)	-10.71	-13.79	3.45

Table 21: Percent Reduction of Average Number of Cache Misses, Cache References, and Average Cache Miss/Reference Ratio over Control by Algorithms 3 and 2 and over Algorithm 2 by Algorithm 3



(a) Percent Reduction of Average Number of Cache Misses, Cache References, and Cache Miss/Reference Ratio over Control for Algorithms 3 and 2

(b) Percent Reduction of Average Number of Cache Misses, Cache References, and Cache Miss/Reference Ratio over Algorithm 2 for Algorithm 3

Figure 23: Percent Reduction of Average Number of Cache Misses, Cache References, and Average Cache Miss/Reference Ratio over Control by Algorithms 3 and 2 and over Algorithm 2 by Algorithm 3

Although there is a negative percent reduction of the average number of cache misses, cache references, and cache miss/reference ratio over control for Algorithm *Cache Miss Priority CPU Scheduler* , this occurs for Algorithm *Priority CPU Scheduler* as well. However, Algorithm *Cache Miss Priority CPU Scheduler* exhibits a positive percent improvement in both performance and power savings when compared to control. This implies that the context switching strategy employed by Algorithm *Cache Miss Priority CPU Scheduler* is an effective means to improving power savings and performance, despite any overhead, when compared to control. However, when comparing Algorithms 3 and 2, not only is there an improvement over Algorithm *Priority CPU Scheduler* in terms of performance and the performance-energy metrics by Algorithm *Cache Miss Priority CPU Scheduler* , Table 21 and Figure 23(b) show that there is also a reduction in the average cache miss to reference ratio. We believe this reduction is the reason why Algorithm *Cache Miss Priority CPU Scheduler* is an effective algorithm for creating both a significant power savings and a performance gain over Algorithms 1 and 2 and over the default Linux CPU scheduler and frequency scaling governor.

9.2.4 Evaluation of Algorithm *Context Switch Priority CPU Scheduler*

Like our earlier evaluations, in our evaluation of Algorithm 4 (*Context Switch Priority CPU Scheduler*), we again assumed that each process in the system has different (not necessarily unique) priorities. Also, we again assigned dynamic priorities to these processes to account for the possibility that a process may change priorities during its execution cycle. These priorities were implemented using Linux nice values, and may or may not have been the default nice value of 0. Also, in our survey, we measured the number of context switches and CPU migrations generated by each process. We measured these values for a task containing twenty-four concurrently executing processes (the same benchmarks as Task 4 in section 9.2.2), and then calculated the average number of context switches and CPU migrations for the entire task. We then compared these values to those for the control case and to Algorithm 2, when using Task 4 with the dynamic priority test trial. The results of this evaluation are summarized in Table 22 and Figure 24 as follows.

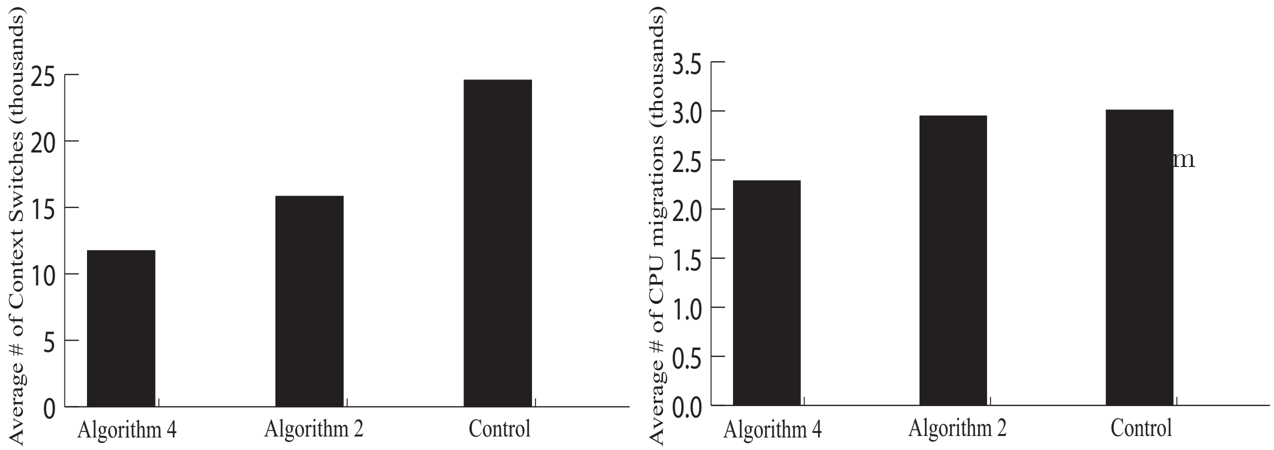
Unlike Algorithm *Priority CPU Scheduler*, Algorithm *Context Switch Priority CPU Scheduler* uses the number of context switches per second of execution time and the number of CPU migrations per second of execution time generated by a process at runtime as one of the determining factors to schedule processes. If a process is context switched too often and consequently changes the CPU upon which it executes often, it will cause an increase in this process's number of context switches per second and number of CPU migrations per second, thereby increasing its context switching overhead. This can have a negative impact upon its performance. To prevent this, Algorithm *Context Switch Priority CPU Scheduler* does not context switch such a process to cores operating at a higher frequency. Thus, if the number of context switches and CPU migrations for an executing process is too high, then a different strategy is implemented by Algorithm *Context Switch Priority CPU Scheduler* . This new strategy's goal is to improve performance by reducing the overhead associated with context switching and CPU migration, while still allowing a sufficient number of CPU migrations of processes to higher frequency cores to maintain power savings.

In our test case, we assigned dynamic priorities to our processes, and in order to test the scenario where Algorithms 1 and 2 exhibited the worst performance (which was Task 4), we used twenty-four concurrently executing processes. Like the dynamic priorities trial type for the first two algorithms, the first priority (Priority 1) was assigned to a process for its first three execution cycles, and its second priority (Priority 2) was assigned during its last two execution cycles. Also, these priorities were unique, with the exception of those assigned to multiple instances of identical benchmarks. Since these priorities were the same as those used for Algorithms 1 and 2, they are shown in Table 6. Again, like the control groups used for Algorithms 1 and 2, our control group consisted of the same processes executing with the default Linux scheduler and CPU frequency scaling governor.

We calculated the average number of context switches and CPU migrations for our task obtained by Algorithm *Context Switch Priority CPU Scheduler* , Algorithm *Priority CPU Scheduler* , and by the control group. These results are shown in Table 22 and Figure 24. We also measured the instructions per second (I/S) for each benchmark and used the average of these values to calculate the instructions per second for the entire task. In addition, we measured the power usage, in watts, that the system used while executing our task. We then used these values to calculate two metrics, (I/S)/watts and (average process completion time) · watts for our task in the test and control groups. Table 23 and Figure 25(a)-(c) depict these results. Also, we calculated the performance gain/loss and power savings of the test versus control group, as well as the percent improvement of both performance metrics obtained by using Algorithm *Context Switch Priority CPU Scheduler* versus the default Linux scheduler and CPU frequency scaling governor. These results are presented in Table 24 and Figure 25(d). Finally, we calculated the percent reduction of the average number of context switches and CPU migrations obtained by Algorithms 4 and 2 when compared to the default Linux scheduler and CPU frequency scaling governor (control group) and that obtained by Algorithm *Context Switch Priority CPU Scheduler* when compared to Algorithm *Priority CPU Scheduler* . These results are shown in Table 25 and Figure 26.

	Algorithm 4	Algorithm 2	Control
Average Number of Context Switches (in thousands)	11.76	15.85	24.59
Average Number of CPU Migrations (in thousands)	2.29	2.95	3.01

Table 22: Average Number of Context Switches and CPU Migrations for Algorithms 4, 2, and Control



(a) Average Number of Context Switches for Algorithms 4, 2, and Control

(b) Average Number of CPU Migrations for Algorithms 4, 2, and Control

Figure 24: Average Number of Context Switches and CPU Migrations for Algorithms 4, 2, and Control for Task 4 (dynamic priorities)

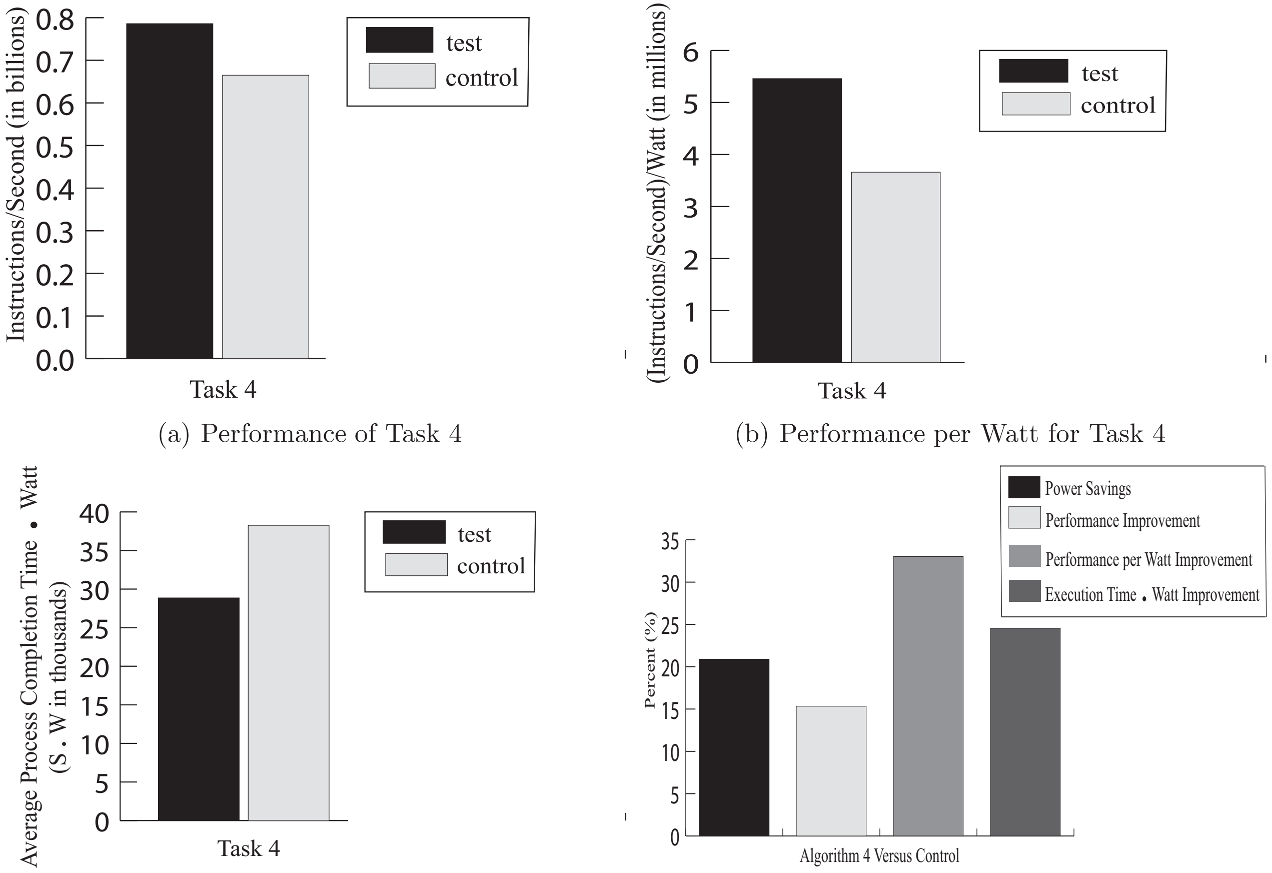
Table 22 and Figure 24 show that the average number of context switches and CPU migrations was lower for Algorithm *Context Switch Priority CPU Scheduler* when compared to that for both Algorithm *Priority CPU Scheduler* and control. This indicates that the strategy employed by Algorithm *Context Switch Priority CPU Scheduler* to use runtime context switch and CPU migration feedback to control the number of context switches is effective. This reduction is achieved because this algorithm does not context switch CPU intensive processes to higher frequency cores if their nice values are less than 9, their performance indices are true, and if their context switch CPU migration indices are true, or if non-CPU intensive processes have nice values less than or equal to -11, if the load average is less than or equal to *num_cpus*, if their performance indices are true, and if their context switch CPU migration indices are true, and if they have nice values greater than -11, the load average is less than 0.15 * *num_cpus*, their performance indices are true, and if their context switch CPU migration indices are true. Since a process's context switch CPU migration index represents its number of context switches and CPU migrations, Algorithm *Context Switch Priority CPU Scheduler* will not context switch processes that generate many context switches nor CPU migrations, thus reducing context switching overhead and improving performance. However, the algorithm does allow a sufficient number of context switching to enable the performance gains achieved when processes execute upon higher frequency cores. This is due to the threshold values present in the context switch CPU migration index and due to the fact that this index is not employed by the algorithm to migrate non-CPU intensive processes to lower frequency cores in a heavily loaded system.

	Task 4	
	test	control
Performance Avg Ins/Sec (in billions)	0.79	0.67
Performance per Watt Avg (Ins/Sec)/Watt (in millions)	5.46	3.66
Execution Time · Watt Avg Time (Sec) · Watt (in thousands)	28.86	38.26

Table 23: Performance, Performance per Watt, and Execution Time · Watt for Task 4 (dynamic priorities)

	Task 4
Total Power Savings (Watts)	38
Power Savings (%)	20.88
Performance Improvement (%)	15.34
Performance per Watt Improvement (%)	33.02
Execution Time · Watt Improvement (%)	24.56

Table 24: Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt by Algorithm 4 for Task 4 (dynamic priorities)



(a) Performance of Task 4

(b) Performance per Watt for Task 4

(c) Execution Time · Watt for Task 4

(d) Percent Power Savings and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt by Algorithm 4 (Task 4)

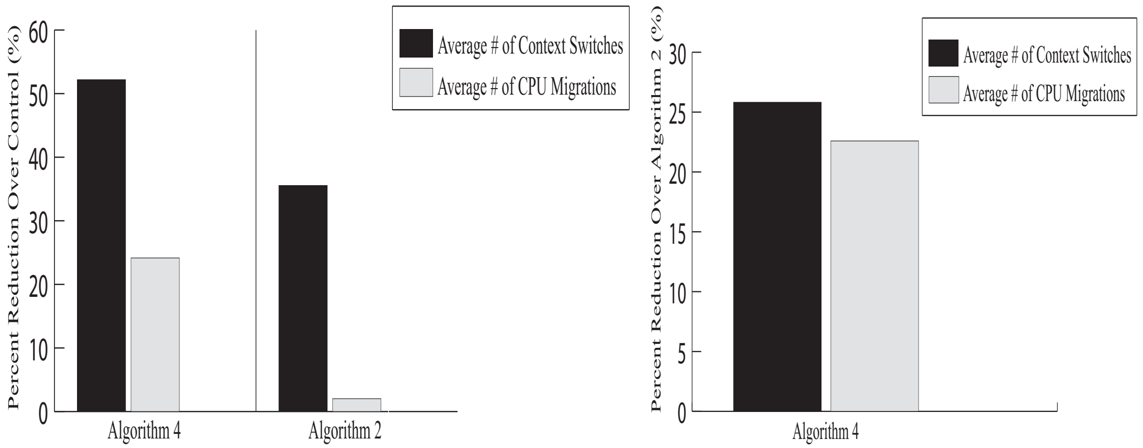
Figure 25: Performance, Performance per Watt, Execution Time · Watt, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt by Algorithm 4 for Task 4 (dynamic priorities)

As shown in Table 23 and Figures 25(a)-(c), performance, in terms of average instructions per second, and performance per watt were both higher for the test group than the control group by Algorithm *Context Switch Priority CPU Scheduler* for Task 4. Also, the metric execution time · watt was lower for the test group than the control group. These differences were even more significant than those seen by Algorithm *Cache Miss Priority CPU Scheduler* . Since a higher value for the first metrics and a lower value for the latter imply a more efficient algorithm, we believe Algorithm *Context Switch Priority CPU Scheduler* is a more efficient algorithm in terms of both power savings and performance than the default Linux CPU frequency scaling governor, and even more so than Algorithm *Cache Miss Priority CPU Scheduler*.

Also, Table 24 and Figure 25(d) show that there is a very significant increase in the percent power savings and percent improvement of performance, performance per watt, and execution time · watt for Algorithm *Context Switch Priority CPU Scheduler* when compared to control. In fact, the power savings was as high as 38 watts while a performance gain of 15.34% was also seen for Task 4. This led to a very significant improvement of the performance per watt metric (33.02%). This indicates that the strategy employed by Algorithm *Context Switch Priority CPU Scheduler* to control context switching overhead is an effective means to increasing power savings while simultaneously improving performance significantly. This strategy is even more effective than that utilized by Algorithm *Cache Miss Priority CPU Scheduler* .

	Percent Reduction Over Control		Percent Reduction Over Algorithm 2
	by Algorithm 4	by Algorithm 2	by Algorithm 4
Average Number of Context Switches (%)	52.19	35.55	25.82
Average Number of CPU Migrations (%)	24.15	2.02	22.59

Table 25: Percent Reduction of Average Number of Context Switches and CPU Migrations over Control by Algorithms 4 and 2 and over Algorithm 2 by Algorithm 4



(a) Percent Reduction of Average Number of Context Switches and CPU Migrations over Control for Algorithms 4 and 2

(b) Percent Reduction of Average Number of Context Switches and CPU Migrations over Algorithm 2 for Algorithm 4

Figure 26: Percent Reduction of Average Number of Context Switches and CPU Migrations over Control by Algorithms 4 and 2 and over Algorithm 2 by Algorithm 4

As shown by Table 25 and Figure 26, there is a greater than 50% reduction in the average number of context switches by Algorithm 4 over control and a very significant reduction in the average number of CPU migrations over control by Algorithm *Context Switch Priority CPU Scheduler* . This reduction is greater than that seen by Algorithm *Priority CPU Scheduler* . Since there is also a very significant improvement in performance seen by Algorithm *Context Switch Priority CPU Scheduler*, this implies that the reduction of context switching overhead induced by Algorithm *Context Switch Priority CPU Scheduler* is high enough to increase performance but is not so high as to stifle the context switching necessary to migrate processes that can benefit from executing upon higher frequency cores. This implies that the context switching strategy used by Algorithm *Context Switch Priority CPU Scheduler* is a very effective means to improving power savings and performance, despite any overhead, when compared to control, and is even more effective than that used by Algorithm *Cache Miss Priority CPU Scheduler* .

In addition, as Table 25 and Figure 26(b) show, there is also a significant reduction in the average number of context switches and CPU migrations generated by Algorithm *Context Switch Priority CPU Scheduler* when compared to Algorithm *Priority CPU Scheduler*, which does not use context switches nor CPU migrations as a scheduling criterion. This difference is even greater than that seen for Algorithm *Cache Miss Priority CPU Scheduler*. We believe this reduction allows Algorithm *Context Switch Priority CPU Scheduler* to achieve both a significant power savings and a significant performance gain over the default Linux CPU scheduler and frequency scaling governor, and even more so than Algorithm *Cache Miss Priority CPU Scheduler* and Algorithms *CPU Scheduler* and *Priority CPU Scheduler*.

CHAPTER 10

CONCLUSION AND FUTURE RESEARCH

The main motivation of our research was to design methods to allocate CPU resources in a heterogeneous multicore processor system with the goal of lowering the global power budget and creating a minimal performance loss (or a performance gain). To accomplish this goal, we used “application-driven” feedback, in which an executing application gave feedback (regarding its performance) to the operating system at runtime, and the operating system, in turn, dynamically scheduled the system’s CPU hardware resources based upon this feedback. We presented four CPU scheduling algorithms that create and utilize cpusets containing varying numbers of cores/processors of varying frequencies, with the goal of lowering the global power budget in a multicore or multiprocessor system, while creating a minimal performance loss, and in some cases, a performance gain. Our algorithms not only match a process to cores better suited to execute this process based upon its performance characteristics, but also consider the performance characteristics of the task as a whole, process priority, the cache miss/cache reference ratio of the executing process, the number of context switches and CPU migrations of the executing process, and the system load. In doing so, the implementation of our algorithms not only results in significant power savings, but for a multicore system containing eight cores executing three, five, eight, or twenty-four concurrently running benchmarks, also results in a significant improvement in performance per watt, execution time · watt, and in some cases, improvement in performance itself. We have also shown, in certain cases, a total power savings of up to 41 watts, or 21.93%, and a performance gain of 15.34%, when using our algorithm compared to using the default Linux scheduler and “on demand” CPU frequency scaling governor.

In addition to this, we have measured the interaction of our algorithm with process priority (nice values). We have shown that our algorithm does not negatively affect Linux priority scheduling, and that process completion order, as dictated by priority values, is not affected, but performance-energy improvements and power savings are obtained.

We believe our work is advantageous over a dynamic voltage and frequency scaling (DVFS) algorithm for the following reasons. Firstly, our algorithms context switch a non-CPU intensive process to higher frequency cores in a lightly loaded system, thus improving performance, whereas a DVFS governor will lower the core's frequency upon which this process is executing, thus degrading performance. Also, our algorithms use "application-driven" feedback to schedule processes upon cores based upon a process's performance characteristics at runtime. Three of our algorithms also consider a process's priority, both static and dynamic, as a scheduling criterion and do not adversely affect process completion order as dictated by process priority. Two of our algorithms also use the cache miss/cache reference ratio and the number of context switches and CPU migrations as scheduling criteria, in an attempt to moderate the number of context switches generated by the process and thus lower context switching overhead and improve performance. Finally, our algorithms avoid large initial load imbalances by using a "throughput estimate", which is a novel method of initial task assignment.

In our research, we have tested tasks composed of multithreaded processes and those containing both CPU intensive and non-CPU intensive processes. We have shown that our CPU scheduling algorithms are superior to the default Linux scheduler and CPU frequency scaling governor in terms of both performance and energy as well as power savings for tasks composed of three, five, eight, and twenty-four concurrently executing benchmarks, without adversely impacting process priority scheduling.

The significance of this research is that in a data center, like the server systems used for cloud computing, even a modest power savings, and certainly a power savings of 41 watts, for each server can result in a huge power savings for the data center as a whole. Also, decreased power dissipation can result in decreased heat dissipation. This can further reduce power usage and increase CPU and hardware component life. Finally, a decrease in the power consumption of CPUs is important for notebook computers, where prolonging battery life is crucial and cooling hardware components may be difficult.

There are many future extensions of this work. An obvious extension is to use a many-core chip, one with tens, hundreds, or even thousands of cores, in our research. Another may be to consider the remaining threads in a process as a performance characteristic. Finally, we may consider different core partitioning strategies, with the goal of spreading heat dissipation and dispersing thermal hot spots or increasing performance.

BIBLIOGRAPHY

[1] T A AlEnawy and H Aydin. Energy-aware task allocation for rate monotonic scheduling. In *11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS '05)*, pages 213–223, San Francisco, CA, USA, March 2005.

[2] G M Almeida, R Busseuil, E A Carara, N Hebert, S Varyani, G Sassatelli, P Benoit, L Torres, and F G Moraes. Predictive dynamic frequency sacaling for multi-processor systems-on-chip. *2011 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1500–1503, May 2011.

[3] Website, 2012. $http://www.amd.com/uk/products/technologies/amd-powernow-technology/Pages/amd-powernow-technology.aspx$.

[4] Website, 2011.
$http://products.amd.com/en-us/OpteronCPUDetail.aspx?id=647$.

[5] M Becchi and P Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. *Journal of Instruction-Level Parallelism*, 10:1–26, 2008.

[6] K Choi, W Lee, R Soma, and M Pedram. Dynamic voltage and frequency scaling under a precise energy model considering variable and fixed components of the system power dissipation. *ICCAD*, pages 29–34, 2004.

[7] M Creeger. Multicore CPUs for the masses. *QUEUE*, September 2005.

[8] Website, 2010. $http://software.intel.com/en-us/articles/$
$detecting-memory-bandwidth-saturation-in-threaded-applications/$.

[9] Website, 2012. $http://en.wikipedia.org/wiki/Dynamic\_voltage\_scaling$.

[10] A Fedorova, J C Saez, D Shelepov, and M Prieto. Maximizing power efficiency with asymmetric multicore systems. *Commun. ACM*, 52(12):48–57, 2009.

[11] A Fedorova, D Vengerov, and D Doucette. Operating system scheduling on heterogeneous core systems. In *Proceedings of 2007 Operating System Support for Heterogeneous Multicore Architectures*, 2007.

[12] M Franklin. Notes from ENEE759M: Microarchitecture. Spring 2008.

[13] S Ghiasi, J Casmira, and D Grunwald. Using IPC variation in workloads with externally specified rates to reduce power consumption. *Workshop on Complexity-Effective Design*, June 2000.

[14] S Ghiasi, T Keller, and F Rawson. Scheduling for heterogeneous processors in server systems. In *Proc. Computing Frontiers*, pages 199–210. ACM Press, 2005.

[15] K Hinum. Core 2 duo notebook processor (CPU). Website, 2011. $http : //www.notebookcheck.net/Intel-Core-2-Duo-Notebook-Processor.7322.0.html$.

[16] Website, 2006.
$http : //www.ibm.com/developerworks/linux/library/$
$1 - scheduler$.

[17] C Isci, A Buyuktosunoglu, C Y Cher, and P Bose anad M Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[18] W Knight. Two heads are better than one. *IEEE Review*, September 2005.

[19] R Kumar, K I Farkas, N P Jouppi, P Ranganathan, and D M Tullsen. A multi-core approach to addressing the energy-complexity problem in microprocessors. *Workshop on Complexity-Effective Design*, June 2003.

[20] R Kumar, K I Farkas, N P Jouppi, P Ranganathan, and D M Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, December 2003.

[21] R Kumar, K I Farkas, N P Jouppi, P Ranganathan, and D M Tullsen. Single-ISA heterogeneous multicore architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[22] J C Mogul, J Mudigonda, N Binkert, P Ranganathan, and V Talwar. Operating systems and asymmetric single-ISA CMPs: The potential for saving energy. Technical report, HPL-2007-140, August 2007.

[23] Website, 2012. $http : //en.wikipedia.org/wiki/Multi - core\_processor$.

[24] Website, 2011. $http : //en.wikipedia.org/wiki/Pentium\_4$.

[25] K K Rangan, G Wei, and D Brooks. Thread motion: fine-grained power management for multi-core systems. *ACM SIGARCH Computer Architecture News*, 37(3), June 2009.

[26] J C Saez, M Prieto, A Fedorova, and S Blagodourov. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of ACM (Eurosys '10)*, April 2010.

[27] B Schauer. Multicore processors - a necessity. Website, 2008. $http : //www.csa.com/discoveryguides/multicore/review.pdf$.

[28] J H Schonherr, J Richling, M Werner, and G Muhl. A scheduling approach for efficient utilization of hardware-driven frequency scaling. pages 367–376, Berlin, Germany, February 2010. VDE Verlag.

[29] Website, 2004. *http : //searchdatacenter.techtarget.com/definition/multi − core − processor*.

[30] D Shelepov and A Fedorova. Scheduling on heterogeneous multicore processors using architectural signatures. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, 2008.

[31] D Shelepov, J C Saez, and et al. S Jeffery. HASS: A scheduler for heterogeneous multicore systems. *ACM Operating System Review*, 43(2), 2009.

[32] R Strong, J Mudigonda, J C Mogul, N Binkert, and D Tullsen. Fast switching of threads between cores. *ACM SIGOPS Operating Systems Review*, 43(2), April 2009.

[33] M A Suleman, O Mutlu, M K Qureshi, and Y Patt. Accelerating critical section execution with asymmetric multicore architectures. *IEEE Micro*, 30(1):60–70, 2010.

[34] Website, 2012. *http : //en.wikipedia.org/wiki/Underclocking*.

[35] A Vajda. Space-shared and frequency-scaling based task scheduler for many-core OS. *Workshop on Power Aware Computing and Systems 2009 (HotPower '09)*, 2009. with SOSP2009.

VITA

Graduate College
University of Nevada, Las Vegas

Rajesh Patel

Home Address:
2001 Shelbyville Street
Henderson, NV 89052

Degrees:
Bachelor of Science, Biological Sciences
University of Southern California

Bachelor of Science, Nuclear Medicine
University of Nevada, Las Vegas

Master of Science, Computer Science
University of Nevada, Las Vegas