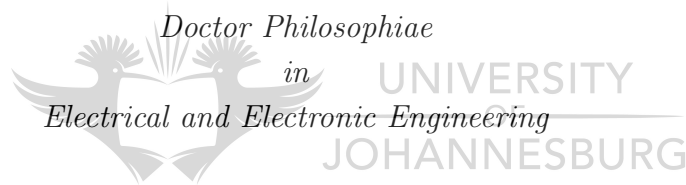


Operating System Scheduling Optimization

By

GEORGE GEORGEVICH ANDERSON

A thesis submitted in fulfilment of the requirements for the degree
of



Faculty of Engineering and the Built Environment
UNIVERSITY OF JOHANNESBURG

Supervisor: Prof. Tshilidzi Marwala
Co-supervisor: Prof. Fulufhelo Vincent Nelwamondo

February 2013

Declaration

I herewith declare that I have produced this thesis without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This thesis has not previously been presented in identical or similar form to any other South African or foreign examination board.

The thesis work was conducted from January 2009 to January 2013 under the supervision of Professor Tshilidzi Marwala and Professor Fulufhelo Vincent Nelwamondo.



George Georgevich Anderson
Johannesburg, South Africa
February 2013

To my parents, George Oppan Anderson and Elena Anderson, for teaching
me how to work hard.



Acknowledgements

I thank God for giving me the ability, creativity, strength, and perseverance to carry out this work. My name appears as the sole author of this dissertation. However, this product would not be possible without the involvement of several people. My supervisor, Professor Tshilidzi Marwala, and my co-supervisor, Professor Fulufhelo Vincent Nelwamondo, guided my research and advised me on which methods to use to tackle the various problems. They were also instrumental in motivating me, and encouraging me, which helped me through some tough times.



Abstract

This thesis explores methods for improving, or optimizing, Operating System (OS) scheduling. We first study the problem of tuning an OS scheduler by setting various parameters, or knobs, made available. This problem has not been addressed extensively in the literature, and has never been solved for the default Linux OS scheduler. We present three methods useful for tuning an Operating System scheduler in order to improve the quality of scheduling leading to better performance for workloads. The first method is based on Response Surface Methodology, the second on the Particle Swarm Optimization (PSO), while the third is based on the Golden Section method. We test our proposed methods using experiments and suitable benchmarks and validate their viability. Results indicate significant gains in execution time for workloads tuned with these methods over execution time for workloads running under schedulers with default, un-optimized tuning parameters. The gains for using RSM-based over default scheduling parameter settings are only limited by the type of workload (how much time it needs to execute); gains of up to 16.48% were obtained, but even more are possible, as described in the thesis. When comparing PSO with Golden Section, PSO produced better scheduling parameter settings, but it took longer to do so, while Golden Section produced slightly worse parameter settings, but much faster. We also study a problem very critical to scheduling on modern Central Processing Units (CPUs). Modern CPUs have multicore designs, which corresponds to having more than one CPU on a single chip. These are known as Chip Multiprocessors (CMPs). The CMP is now the standard type of CPU for many different types of computers, including Personal Computers. The architecture of the CMP makes it

important for an Operating System scheduler to take into account competition for processor-related resources by the various tasks i.e. resource contention. Therefore it is important to make a good decision when choosing tasks to run on closely-related cores of a CMP. Scheduling for such architectures remains a challenge because of resource contention, which results in degraded performance for tasks using the same types of resources and running on cores sharing these hardware resources, such as the LLC (Last Level Cache), memory bus, memory controller, and prefetch hardware. We solve this problem by implementing a theoretically optimal scheduler based on Integer Programming, and using Machine Learning models to learn how this optimal scheduler schedules tasks. We develop a black box scheduler, which learns how to map task attributes, such as MPI (Misses Per Instruction), to scheduling decisions. We name our scheduling approach Black Box Learned Optimization. Black Box Learned Optimization produces results almost the same as with the optimal scheduler in most cases tested, but with significant gains in execution time. We implemented Multilayer Perceptron (MLP) and Support Vector Machine (SVM) schedulers using Learned Optimization, in addition to other models. SVM performed best with the nine workloads tested; approximately equal to the optimal scheduler in seven out of nine workloads, with faster execution time. Our SVM scheduler is also better than the state-of-the-art scheduler in eight out of nine workloads.

Signature Page

1. Supervisor: Prof. Tshilidzi Marwala

Supervisor's signature:

2. Co-supervisor: Prof. Fulufhelo Vincent Nelwamondo

Co-Supervisor's signature:



UNIVERSITY
OF
JOHANNESBURG

Contents

List of Figures	xii
List of Tables	xiii
List of algorithms	xiv
Glossary	xv
Author's Publications from this Thesis	xvii
1 Introduction	1
1.1 Problem Description	1
1.1.1 Operating Systems Background	1
1.1.2 Operating System Scheduler Tuning	2
1.1.3 Multicore Scheduling	2
1.2 Goals and Objectives	5
1.3 Contributions	6
1.4 Organization of the Thesis	7
2 Literature Review	8
2.1 Operating System Overview	8
2.2 Operating System Scheduler Tuning	9
2.3 Chip Multiprocessors	12
2.4 Operating System Scheduler Processor Allocation	13
2.5 Machine Learning in Scheduling	16
2.6 Summary	18

3	Experimental Approach	19
3.1	Approach Overview	19
3.2	The LinSched Simulator: Comparison of Bootstrapping and Finite State Machine Simulations of a Scheduling Benchmark	21
3.2.1	Overview	21
3.2.2	Need for LinSched and Simulated Benchmarks	22
3.2.3	Related Work	23
3.2.4	Process Scheduling in the Linux Operating System	23
3.2.5	The LinSched Simulator	24
3.2.6	Benchmark Simulation	24
3.2.6.1	Finite State Machine Approach	26
3.2.6.2	Bootstrapping	27
3.2.7	Results	28
3.2.8	Conclusion	30
3.3	AKULA Multicore Scheduling Simulator	31
3.3.1	Overview	31
3.3.2	Validation	31
3.3.3	AKULA Framework	31
3.4	Benchmarks	32
3.5	Experimental Platforms	33
3.6	Summary	33
4	Operating System Scheduler Tuning Using Response Surface Method- ology	34
4.1	Summary	34
4.2	Scheduler Tuning Introduction	35
4.3	Response Surface Methodology Overview	36
4.3.1	Design of Experiments	36
4.3.2	Response Surface Methodology	38
4.4	Use of Response Surface Methodology and Other Statistical Methods in Software Tuning	39
4.5	The Linux Scheduler	41
4.6	Scheduler Tuning Experiment Design	42

4.7	Scheduler Tuning Model Analysis	43
4.8	Experiment and Results	48
4.8.1	Benchmark Experiment	50
4.8.2	Results of Experiment (Optimization)	52
4.9	RSM-Based Parameter Tuning Methodology	53
4.10	Conclusion	54
5	Application of Standard Optimization Methods to Operating System Scheduler Tuning	56
5.1	Summary	56
5.2	Scheduler Tuning Introduction	57
5.3	Optimization in Software Systems	58
5.4	Operating System Scheduling in Linux	59
5.5	Operating System Scheduler Simulation	59
5.6	Optimization of the Operating System Scheduler	61
5.6.1	Global Optimization	62
5.6.2	Local Optimization	63
5.7	Results	66
5.8	Standard Optimization-Based Parameter Tuning Methodology	68
5.9	Conclusion	69
6	Black Box Learned Optimization Scheduling	70
6.1	Summary	70
6.2	Processor Allocation Introduction	71
6.3	Related Work Summary	73
6.4	The Optimal Scheduling Algorithm and a Relatively Near-Optimal Scheduling Algorithm	76
6.4.1	Optimal Scheduling	76
6.4.2	Relatively Near-Optimal Scheduling	76
6.5	Implementation Environment	77
6.6	A Linear Programming-Based Multicore Scheduler	78
6.7	Learned Optimization	79
6.8	MLP and SVM	80
6.8.1	MLP	80

CONTENTS

6.8.2	SVM	83
6.9	MLP and SVM-Based Multicore Schedulers	86
6.10	Results	87
6.10.1	Workloads	87
6.10.2	Schedulers	87
6.10.3	Significance of Results	92
6.11	Conclusion	93
7	Conclusions	95
7.1	Achievements	95
7.2	Future Work	97
7.3	Summary	97
	References	98



List of Figures

1.1	A schematic view of a multicore architecture. The package (CPU) has 2 chips, with a total of 4 cores (C0 ... C3). Each of the 2 chips has 2 cores sharing a LLC cache. The prefetchers are shown, as well as the FSB (Front Side Bus), which is shared by the 2 chips and connects the chips to the memory controller, which is connected to the memory. . . .	4
3.1	Finite State Machine for Sender	26
3.2	Finite State Machine for Receiver	27
3.3	Regression Model for FSM. Runtime in milliseconds vs. FSM Jiffies . . .	29
3.4	Regression Model for Bootstrapping. Runtime in Milliseconds vs. Bootstrapping Jiffies	30
4.1	Central Composite Design (CCD) for three factors, with stars on axes. Circles and stars show possible settings for the factors.	39
4.2	Box-Behnken Design (BBD) for three factors. Circles show possible values for the factors.	39
4.3	Box-Cox plot for model	44
4.4	Response cube	45
4.5	Normal plot of residuals	45
4.6	Residuals versus predicted plot	46
4.7	Residuals versus run number plot	47
4.8	Residuals versus Latency plot	48
4.9	Residuals versus Minimum Granularity plot	49
4.10	Residuals versus Wakeup Granularity plot	49

LIST OF FIGURES

4.11	3D response surface for runtime (latency is constant with a value of 500050000)	54
4.12	Hackbench turnaround time for the optimized and unoptimized Scheduler	54
5.1	Linear Relationship Between LinSched Simulator Jiffies Metric and Hackbench Seconds Metric	61
5.2	Minimum turnaround times achieved	67
5.3	Minimum iterations needed to achieve target turnaround	67
5.4	Minimum turnaround times achieved for a range of workloads	68
5.5	Minimum iterations needed to achieve target turnaround, for a range of workloads	68
6.1	An Assignment Problem Graph: Vertices A-H are the tasks, each of which has to be assigned to one of two sets of cores, I and J. Each edge has a weight, which is the cost of assignment to a particular set of cores. In the model used in this thesis, the cost is also dependent on other assignments.	79
6.2	Multilayer Perceptron. The input, hidden, and output variables are represented by nodes, and the weights are represented by links between nodes. x_0 and z_0 are bias nodes. The arrows show the direction of information flow through the network.	83
6.3	Transformation of data to aid in separation [1].	84
6.4	Average run times for 5 of the approaches we studied in <i>ms</i>	92

List of Tables

3.1	AKULA Classes and Modifications Made for this Thesis	32
4.1	Model lack of fit summary	46
4.2	ANOVA for response surface quadratic model	50
5.1	Meta-optimization: RSM-designed PSO experiment runs	63
6.1	SPEC CPU2006 benchmark descriptions. Type I is Integer, F is Floating Point.	88
6.2	Workload Composition	89
6.3	Average degradation and unfairness performance figures for various workloads and Optimal, DI, ModLP, MLP1, MLP2, and SVM Schedulers (Best Figure is Bold; Average Degradation not compared with Optimal)	89
6.4	Average degradation percentage improvement for DI, ModLP, MLP2, and SVM, for all Workloads	89

List of Algorithms

4.1	RSM-based operating system scheduler tuning methodology	55
5.1	Simple Scheduling Algorithm	58
5.2	Parameter discovery algorithm 1, used by Golden Section 1	65
5.3	Parameter discovery algorithm 2, used by Golden Section 2	65
5.4	Standard optimization-based operating system scheduler tuning methodology	68
6.1	General Black Box Learned Optimization Algorithm. Instances input into the optimization algorithm are independent variable values.	80
6.2	Simulation main module	86
6.3	Simulation bootstrap module	87
6.4	Simulation scheduler creation module	87
6.5	MLP and SVM Scheduler Development	88

Glossary

AKULA	Java-based multicore scheduling simulator	libquantum	Quantum computer simulation benchmark
API	Application Programming Interface	LibSVM	Support Vector Machine library and tools
CFS	Completely Fair Scheduler	LinSched	Linux scheduling simulator using user-level Linux kernel code
CMP	Chip Multiprocessor; multicore CPU	LP	Linear Programming
CPI	Cycles Per Instruction	M5P	Model 5 Prime decision tree
CPU	Central Processing Unit; the main processor of a computer	mcf	Single-depot vehicle scheduling in public mass transportation benchmark
DI	Distributed Intensity	milc	MIMD Lattice Computation benchmark, simulating lattice gauge theory
FSM	Finite State Machine	MLP	Multilayer Perceptron; a machine learning model based on Neural Networks
GA	Genetic Algorithms	MPI	Misses Per Instruction
gamess	Quantum chemical computation benchmark	ms	milliseconds
gcc	GNU C Compiler benchmark	namd	Benchmark simulating large biomolecular systems
gobmk	Go-playing and analyser benchmark	NetBeans	Java-based Integrated Development Environment
I/O	Input/Output; activities or devices related to data transfer into or out of a computer (main memory)	ns	nanoseconds
IDE	Integrated Development Environment	OS	Operating System; system software responsible for providing an interface to computer resources, and for managing such resources
IP	Integer Programming	PA	Processor Allocation; assignment of tasks to processing cores by the OS
IPC	Instructions Per Cycle	povray	Persistence of Vision ray-tracer benchmark
Jiffy	Linux scheduling time unit, which could be equivalent to 1 millisecond	PSO	Particle Swarm Optimization; global optimization method
lbm	Benchmark simulating incompressible fluids using Lattice Boltzmann Method	REPTree	Reduced Error Pruned Tree
		RSM	Response Surface Methodology
		SLA	Stochastic Learning Automaton

GLOSSARY

SPEC CPU2006	Standard Performance Evaluation Corporation CPU benchmark suite	V/f	Voltage and frequency
SVM	Support Vector Machine; machine learning model	Weka	Waikato Environment for Knowledge Analysis



Author's Publications from this Thesis

- [1] G. ANDERSON, T. MARWALA, AND F.V. NELWAMONDO. **A Response Surface Methodology Approach to Operating System Scheduler Tuning.** In *Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on*, pages 2684–2689, Oct. 2010. Some related content appears in Chapter 4. This paper reports on our study on the use of RSM for scheduler tuning.
- [2] G. ANDERSON, T. MARWALA, AND F.V. NELWAMONDO. **Comparison of Bootstrapping and Finite State Machine Simulations of a Scheduling Benchmark.** In TAREK SOBH AND KHALED ELLEITHY, editors, *Emerging Trends in Computing, Informatics, Systems Sciences, and Engineering*, **151** of *Lecture Notes in Electrical Engineering*. Springer, Nov. 2012. Some related content appears in Chapter 3. This paper reports on our study on bootstrapping and Finite State Machine simulations of scheduling benchmarks, including a comparison.
- [3] G. ANDERSON, T. MARWALA, AND F.V. NELWAMONDO. **Application of Global and One-Dimensional Local Optimization to Operating System Scheduler Tuning.** In *Proceedings of the Twenty-First Annual Symposium of the Pattern Recognition Association of South Africa*, Stellenbosch, South Africa, Nov. 22–23 2010. Some related content appears in Chapter 5. This paper reports on our study on using standard optimization methods for scheduler tuning.
- [4] G. ANDERSON, T. MARWALA, AND F.V. NELWAMONDO. **Multicore Scheduling Based on Learning from Optimization Models.** *International Journal of Innovative Computing, Information and Control*, 2013. To Appear. Some related

AUTHOR'S PUBLICATIONS FROM THIS THESIS

content appears in Chapter 6. This paper reports on our study on Black Box Learned Optimization.



Chapter 1

Introduction

1.1 Problem Description

1.1.1 Operating Systems Background

The Operating System is a very important piece of software present in virtually all computer systems [2, 3, 4]. It belongs to a category of software called system software. The job of system software is to support the use of computer systems, including running of programs, which belong to the other category of software: applications. The more specific job of the operating system is to manage the resources of the computer system. These resources including all components in the computer system that are used. They include main memory, input devices, output devices, storage devices, programs, files, the Central Processing Unit (CPU), etc. There are many types of resources, working together with complex interactions, therefore there has to be a single entity that governs these interactions and uses of resources; this entity is the operating system.

The operating system has many components which work together to help carry out resource management duties [2, 3, 4]. For example there is the memory manager (manages use of main memory), various device drivers (carry out low-level interactions with hardware devices), Input/Output manager (handles data transfer between memory and I/O devices), process manager (creates and destroys processes), file system manager (manages file systems), etc. Another component is the process scheduler, also known as the CPU scheduler, which is the focus of this thesis. In this thesis we use the term *scheduler* to mean CPU scheduler, not to be confused with other schedulers in the operating system, such as the I/O scheduler.

1.1.2 Operating System Scheduler Tuning

The job of the scheduler is to assign tasks to CPUs. Tasks could refer to processes or threads. A process is a program currently running in a computer; in other words, a process is an instance of a program that is being executed while a thread is a flow of execution within a process [75]. A task refers to a process in some operating systems, and to a thread in others. Algorithm 5.1 describes the basic operation of a simple scheduler. The operating system has to reserve some address space in main memory for each task, and maintain some data about each task, such as the execution environment, which includes the state of CPU registers and memory structure. When there are less CPUs in a computer than tasks ready to use a CPU, the scheduler has to decide which task will use a particular CPU next and for how long. This is time scheduling and is carried out by a component of the scheduler called the Job Scheduler [5]. The operating system scheduler may be invoked several times a second to make scheduling decisions, as well as when certain events happen, such as a task finishing its execution [2, 3, 4]. Operating system performance parameters, including scheduler parameters, are set in an iterative cycle of tuning and measuring, which is time consuming [6, 7]. Schedulers, for example, the one in the Linux operating system kernel, have various parameters that are set with no standard methodology in place for doing so. It is therefore very important to develop an appropriate methodology, in order to take the guess work and gut feelings out of the process. There have been other efforts in the past aimed at addressing operating system scheduler tuning, for example the study carried out by Moilanen and Williams [8], but these involve the design of a system which requires the scheduler to be modified, and is specific to that scheduler. Our research results in methodologies for scheduler tuning using optimization methods, which do not require modification of the scheduler and can easily be applied to various schedulers which allow for parameter tuning.

1.1.3 Multicore Scheduling

The scheduler may also carry out space scheduling, where it decides which task will use which CPU. This is critical in multiprocessing systems, and modern multicore systems. The component of the scheduler which does this is called the Processor Allocator (PA) [5]. Multiprocessing systems have more than one CPU. Each CPU is a separate chip

on the motherboard. Such designs are expensive. Modern computers have multicore CPUs. In a simple configuration, there is one physical chip (package), which contains two or more CPU cores. Each core is equivalent to a CPU. Therefore when a multicore CPU with two cores is being used on a computer, two CPUs will be identified. In the old days CPU performance was improved regularly by increasing the clock speed. Moore's law accurately predicted the doubling of transistors on microprocessors every two years [9]. This also has implications on exponential increase in processing speed. Processor designers used advanced techniques such as superscalar issue and instruction pipelining, however, such designs used a huge amount of power, and generated a lot of heat [10]. While performance was increasing exponentially, heating could not. Therefore, the route now taken to better performance is more processing cores per physical chip, allowing more tasks to be executed in parallel [10]. Such CPUs are called multicore CPUs or CMPs (Chip Multiprocessors).

With the advent of multicore systems, the processing elements (cores) share some resources, for example the LLC (last level cache), memory controller, memory bus, and prefetch hardware. Therefore when two or more tasks are running on cores that share these resources, there is resource contention; their performance will be degraded, compared to if each task were to run on its own [11, 12]. Generally, on multicore architectures, competing tasks face performance degradation when co-scheduled together, on separate cores sharing resources, while cooperating (communicating) tasks face performance degradation when not co-scheduled [13]. Because of this degradation, it is important that when the scheduler is making its decisions on which task should run on which core, it takes resource contention into account. Figure 1.1 shows the architecture of a multicore CPU.

Siddha et al. [14], while Senior Engineers at Intel, the largest computer chip manufacturer, emphasized the importance of moving to multicore architectures. They highlighted the importance of scheduling in taking full advantage of multicore architectures, including the need to identify and predict task needs and minimize resource contention i.e. the need for contention-aware scheduling. Without contention aware scheduling, schedulers cannot make good decisions. The common approach nowadays to multicore scheduling in commercial operating systems is load balancing, but this has far from optimal results, because it does not take contention characteristics of various tasks when

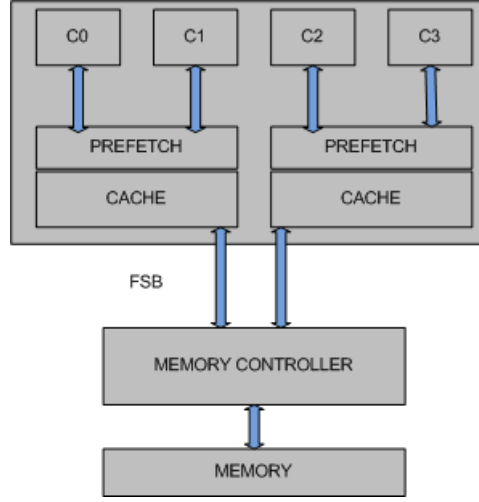


Figure 1.1: A schematic view of a multicore architecture. The package (CPU) has 2 chips, with a total of 4 cores (C0 . . . C3). Each of the 2 chips has 2 cores sharing a LLC cache. The prefetchers are shown, as well as the FSB (Front Side Bus), which is shared by the 2 chips and connects the chips to the memory controller, which is connected to the memory.

co-scheduled into account [15]. Contention-aware scheduling studies fall into two categories: those targeted at on-line scheduling, where scheduling decisions must be made while tasks are running, so the scheduler must be very fast; and optimal scheduling, where the scheduler is not used on-line, but tries to come up with optimal schedules for the purpose of knowing the best possible schedule, in order to help develop scheduling algorithms [16]. There have been studies carried out into optimal contention-aware scheduling, but not many; optimal algorithms have been produced as well as heuristic approximations to the optimal algorithms [16, 17, 18]. However, these algorithms require a priori knowledge of performance degradation of tasks when running in various combinations on cores and are relatively slow; any performance improvement in the execution time of such schedulers increases their usefulness [19, 20]. In this work, we achieve the merging of optimal contention-aware scheduling algorithms and on-line contention aware scheduling, in such a manner that near-optimal, and sometimes optimal results are achieved with relatively short execution time. Also, the heuristic algorithms do not produce schedules as good as the optimal algorithms. The current state-of-the-art contention-aware scheduler is the Distributed Intensity (DI) scheduler [11]. It has the best performance (execution time spent by the scheduler) when gen-

erating schedules, however the schedules it produces are not very close to the optimal schedules (most about 10% worse, with our benchmarks from the SPEC CPU2006 benchmark suite [59]). *We combine the on-line and optimal scheduling approaches by developing a scheduler which targets generating optimal schedules while exhibiting fast execution time.* A thorough literature review has revealed that this approach has not been used before for contention-aware schedulers. We close the gap between optimal schedulers and approximations, by implementing and evaluating optimization and machine learning-based schedulers.

1.2 Goals and Objectives

The goal of this thesis is to study the application of optimization and machine learning techniques in improving the performance of operating system schedulers. How the bottleneck device, such as the CPU, is scheduled, has a huge impact on the performance of the computer system as a whole [21]. The operating system schedulers in question include the ones in the Linux 2.6.23 and 2.6.28 kernels, and generic ones implemented in the AKULA [12, 19, 22] scheduling simulator. Improvement could involve the schedulers in question making better scheduling decisions, so workload throughput is improved, and also the schedulers spending less time in making their decisions. The following objectives have to be met:

- A methodology to tune an operating system scheduler for a specific workload should be formulated. This methodology should be based on experimenting with different scheduler parameters to find a combination that works, in such a manner that the number of experiments, and thus experimentation time, is kept to a minimum.
- An additional methodology to tune an operating system scheduler for a specific workload should be formulated. This additional methodology should be based on standard optimization methods and should consider the operating system's internal design.
- A design for a scheduler, which can optimize the assignment of tasks to cores on a multicore computer system, should be formulated. The scheduling decision-

making time should beat the optimal solution, while the quality of schedules should approach the optimal solution.

1.3 Contributions

The overall work of this thesis involves application of Experimental Statistics, Optimization, and Machine Learning, to an operating system CPU scheduling problem. The following contributions are highlighted here:

- a. A methodology based on Statistical Experiment design has been developed to tune an operating system scheduler for a specific workload. To the best of our knowledge, this has never been studied before. This methodology helps to solve the problem of what values to use for various scheduler parameters. The parameters have wide ranges so using trial and error would require very many cycles of setting parameters, running benchmarks, and evaluating results. Our methodology is based on Response Surface Methodology and keeps the number of experiments (benchmark runs) to a minimum. Results show much better benchmark run times than without the scheduler tuning.
- b. A methodology based on global optimization techniques has been formulated to tune an operating system scheduler. To the best of our knowledge, this approach has never been studied before. Particle Swarm Optimization was used to tune an operating system scheduler, with results showing much better benchmark run times than without scheduler tuning.
- c. A methodology based on one-dimensional optimization techniques has been formulated to tune an operating system scheduler. Our methodology makes novel use of relationships within the scheduler to achieve very good results. To the best of our knowledge, this approach has never been studied before. Golden Section method used. Results show much better benchmark run times than without scheduler tuning.
- d. Methodologies in contributions (b) and (c) above have been compared, and their advantages and disadvantages highlighted.

- e. A novel method for simulating benchmarks in a scheduling simulator was developed: Finite State Machine simulation. This was compared with another state-of-the-art method called bootstrapping [12]. The approach of benchmark simulation was used to evaluate workloads in the global and one-dimensional optimization approaches.
- f. A scheduler focused on multicore scheduling problems was designed, which is faster than the optimal scheduler, while generating schedules which are relatively close to the optimal, outperforming the state-of-the-art contention-aware scheduler [11]. Our scheduler relies on machine learning models. Various such models were evaluated in order to achieve the results.

1.4 Organization of the Thesis

The rest of this thesis is organized as follows. Chapter 2 has the background/literature review; Chapter 3 describes our methodology; Chapter 4 discusses the Response Surface Methodology approach to operating system scheduler tuning; Chapter 5 discusses optimization approaches to operating system scheduler tuning; Chapter 6 discusses task assignment to cores and the Black Box Learned Optimization solution; Chapter 7 has our conclusion.

Chapter 2

Literature Review

This chapter provides background on literature relevant to the problems and solutions being investigated in our research. Section 2.1 gives an overview of operating systems; Section 2.2 discusses work done on tuning of OS schedulers and other software systems; Section 2.3 discusses Chip Multiprocessors (CMPs); Section 2.4 discusses solutions to the OS scheduler Processor Allocation problem, involving allocation of tasks to CMP cores; Section 2.5 discusses the use of Machine Learning in scheduling; Section 2.6 summarizes this chapter.

2.1 Operating System Overview

Modern day computers consist of many different components, such as the CPU, memory, hard disk drives, keyboards, mouse devices and other input devices, monitors, etc. It is important to have suitable interfaces to all these components such that programmers developing software do not have to worry about low level programming details of these device, such as which device registers need to be loaded with which values; the operating system has device drivers which handle such low level details, and these are accessible through system calls by programs. The operating system is therefore an interface used to access various resources. It is also important for there to be a single entity managing these components and ensuring they are used optimally. The operating system also plays this role: a resource manager [4].

The operating system has many components which work together to help carry out resource management duties [2, 3, 4]. For example there is the memory manager

(manages use of main memory by programs), various device drivers (carry out low-level interactions with hardware devices), Input/Output manager (handles data transfer between memory and I/O devices), process manager (creates and destroys processes), file system manager (manages file systems), etc. Another component is the process scheduler, also known as the CPU scheduler, which is the focus in this thesis. We use the term *scheduler* to mean CPU scheduler, not to be confused with other schedulers in the operating system, such as the I/O scheduler.

2.2 Operating System Scheduler Tuning

Schedulers, for example the one in the Linux operating system kernel, have various parameters that are set with little theoretical foundation backing these settings, as evidenced in the lack of literature addressing this issue. Operating system performance parameters, including scheduler parameters, are set in an iterative cycle of tuning and measuring [6, 7]. This involves measuring components of the system such as CPU utilization, memory usage, and throughput. This iterative process consumes a lot of time, involving repetitive phases of measuring and tuning [8, 23, 24]. It would be of great benefit to system engineers and administrators to have a tool that could help to tune parts of an operating system optimally. In this thesis we focus on tuning the operating system scheduler, using the Linux scheduler to demonstrate a proof of concept.

Jain [25] describes performance tuning (adjusting system parameters to optimize performance) as falling under the area of capacity management. Capacity management deals with ensuring current computer resources provide the best possible performance. According to Jain, performance tuning involves the following steps:

- i. Instrument the system. This could involve modifying software so it can be monitored. For example, after every n iterations, the software will write system state to a log file. Or after every n iterations it could call another monitoring program.
- ii. Monitor system usage
- iii. Characterize workload
- iv. Predict performance under different alternatives

- v. Select the lowest cost, highest performance alternative

Scientists used the OFAT (“One-Factor-At-a-Time”) approach for years to develop empirical models linking factors with responses [26]. In a system with n factors, this involved keeping $n - 1$ factors constant and varying the n^{th} factor to measure its effect on the response. The problem with this approach is that it does not take into account the impact of interactions between factors on the response. A factor may not affect the response much on its own, but when combined with another factor, the impact could be significant.

Response Surface Methodology (RSM) has been used in various studies for optimization tasks, though not for the operating system scheduler. Totaro [26] used RSM to configure multi-hop wireless mesh networks. Such networks are set up in emergency situations where wireline networks are not feasible. Their performance metrics (measured by throughput, delay, jitter, and packet delivery ratio) depend on a number of factors such as *average number of neighbours* (network density), *traffic load* (percentage of nodes acting as source traffic generators), *node mobility* (average node speed), *network size* (number of nodes in the system), and the *medium access control protocol*. Totaro developed empirical models relating responses with factors and discovered how significant factors should be configured to optimize the response. He first of all found out whether Design of Experiments (DoE) could be useful in analyzing network system and protocol performance. He used a limited set of factors for this phase. He then used an “expanded design” phase where he used a larger factor space with a fractional factorial design, limiting the number of experiments required, and eliminating factors that have little impact on the response. RSM was used where first order models proved inadequate for optimizing the response.

Vadde [27] presented a methodology to optimize the performance of the layered network protocol stack. The various layers have parameters that can be set but it is difficult to know how to combine settings for several layers together to achieve optimum performance. Vadde used Design of Experiments (DoE) to learn about the various factors and find out which ones are more significant, followed by RSM to optimize the settings for factors in different layers. The protocol stack studied was one used in mobile ad-hoc networks. Our study is similar in that it provides a tool for performance optimization based on RSM, but it is different in that we are focusing on the operating system scheduler, not networks.

2.2 Operating System Scheduler Tuning

Shivam [28] developed a framework for managing benchmarking of systems based on RSM. They used an NFS (Network File System) server to demonstrate the usefulness of their approach. They cite the usefulness of RSM when applied to measuring system performance dependent on various configurations and workloads, listing its ability to “evaluate design and cost trade-offs, explore the interactions of workloads and system choices, and identify optima, crossover points, break-even points, or the bounds of the effective operating range for particular design choices and configurations” as big advantages. They tested their framework by seeing how fast it can find the saturation point for the NFS server i.e. the point at which the response time exceeds a threshold, meaning that this point represents the maximum load the server can handle. Some previous approaches used a *Strawman Linear Search*. This works by incrementing the load, λ , and running t runs for a time period of r for each λ . These increments continue until a saturation point is reached. With this process, if the increments are too small (and the starting load level is far from the saturation point), the benchmarking process will take a long time. If the increments are too large, this approach may overshoot the saturation point. The paper also discusses some older incremental improvements on this approach. With the improved RSM-based approach, RSM is used to select combinations of levels of different factors (workload, physical resources and software configuration) to speed up the convergence to the saturation point. The parameters, such as r and t are chosen so that the response is measured with a given level of confidence, and accuracy.

Sullivan et al. [29, 30] developed a system for tuning software systems using influence diagrams. These diagrams are directed acyclic graphs that capture decision making, performance random variables, and decision maker’s utility. Decision making represents the knobs that are manipulated to tune the software system. The decision maker’s utility represents the performance measures to be optimized. The performance random variables represent the workload characteristics. Their approach involves the use of a workload generator in order to gather parameters of the influence diagrams (random variables are determined using maximum likelihood estimation). These can then be incorporated into the model for a particular system, and incorporated into its implementation. The knob settings are then optimized for the workload.

We are proposing a tool based on Response Surface Methodology (RSM) that could be used to tune an operating system scheduler to give optimum performance for certain

workloads. Such a tool will help to back up parameter-setting with a sound foundation, based on a statistical approach, which will generate a set of parameter settings to test the scheduler with, and then decide on the parameter settings that give an optimal response. The response we are concerned with is the turnaround time of a workload i.e. how fast the workload completes execution.

2.3 Chip Multiprocessors

Microprocessors power all modern-day computers and their performance has been increasing exponentially. Moore's law accurately predicted the doubling of transistors on microprocessors every two years [9]. This also has implications on exponential increase in processing performance, even faster than Moore's law, because with more transistors processor designers have been able to extract more parallelism from software [31]. Processors use *pipelining* to break instructions into larger numbers of smaller steps, reducing the amount of logic that needs to switch in each clock cycle, allowing for an increase in clock rate [10]. The higher the clock rate the faster the processor. Another approach to achieving better performance is *superscalar* execution, which examines instruction streams and chooses instructions that can be executed in parallel, often out of order with respect to the sequences in their programs [10]. These performance improvement approaches are invisible to programmers, because they are given the illusion that instructions are being executed sequentially and in order, instead of overlapped and out of order. However, superscalar processors have limits because instruction streams have limited parallelism, and building such processors is very expensive because of the highly complex logic required for high levels of parallelism. Pipelining also has its limits in terms of how small the pipeline stages can be, because they may be too small to perform any significant logic. Furthermore, significant advances in pipelining and superscalar issue require huge amounts of transistors to be incorporated into the processor, which requires large numbers of engineers to design and verify the processors.

However, the methods for performance improvement described above have reached an electrical power limit. Pipelining and superscalar issue increased processor power requirements from less than one Watt to over 100 Watts from 1985 to 2005 [10]. Cooling technology has not kept up with this dramatic increase in power demand, meaning that if such power requirement trends continue, processors would require sophisticated water

cooling, which would be too expensive for most users. It therefore became important for processor designers to find new ways of harnessing the increasing transistor budget in ways which minimize power requirements and design complexity.

In order to address the problems in the previous paragraph, modern CPUs are now designed with multiple cores (processors) on a single die. These are called Chip Multiprocessors (CMPs). Compared to conventional multiprocessing systems which have two or more CPUs on separate physical chips on the motherboard, performance per unit volume is increased for CMPs. There are also reduced power requirements because processors on the die share certain resources, such as connections to the rest of the system. Cores may also share other resources, such as cache, which enables processes running on different cores to communicate without off-chip access [10]. Thus, performance of interactive workloads (latency-sensitive, such as on desktops) can be improved. CMPs are naturally very well suited to high throughput workloads as found in server environments.

With the advent of CMPs, the processing elements (cores) share some resources, for example the LLC (last level cache), memory controller, memory bus, and prefetch hardware. This leads to some problems. For example, when two or more tasks are running on cores that share these resources, there is resource contention; their performance will be degraded, compared to if each task were to run on its own [11, 12]. This problem becomes more critical as the number of cores per chip increases; 8 to 16 cores per chip are commercially available, for example in the Intel E7-8800 [32], IBM POWER7 [33], and AMD Opteron 6000 CPUs [34]. Intel is developing a CMP with more than 50 cores on a chip [35]. Hence there is a need for contention-aware scheduling by the operating system; for the Processor Allocator to take into account the architecture of the CMP when assigning tasks to cores. This drives the need for the research reported in this thesis.

2.4 Operating System Scheduler Processor Allocation

Illikkal et al. [36] developed a rate-based approach to resource management of CMPs. They make use of rate-based Quality of Service (QoS) methods to decrease resource contention. They make use of hardware Voltage and Frequency (V/f) Scaling and Clock Modulation/Gating. V/f involves changing the frequency of a core in order to reduce

power consumption, while Clock Modulation involves feeding the clock to the processor for short durations. The core is active only for these durations. The time the core is halted translates to reductions in cache space and memory bandwidth requirements of the task running on the core. Tasks on the other cores then experience less contention for the shared resources. These could have higher priority, for example, hence the need for reduced contention. Illickal et al.'s results showed their approach to be flexible and effective at ensuring QoS. Our multicore scheduling work focuses on assigning cores to tasks in such a way as to reduce performance degradation due to resource contention.

Hoh [37] made use of a vector-based approach to co-schedule tasks which were dissimilar in their use of CPU resources. Each task had a vector associated with it, which included elements for various CPU performance counters, such as number of all instructions completed, number of floating point instructions, number of special (e.g. graphic) instructions, number of L2 cache cycles, etc. When selecting a task, the task with the greatest vector difference from the average is selected. Their approach was integrated into the Linux Kernel 2.6.26 Completely Fair Scheduler. Performance tests showed that Hoh's approach was slightly worse than the vanilla Linux kernel when only 2 tasks were running, due to his method's overhead, while it was much better than the vanilla kernel when more than 2 tasks were running. Our multicore scheduling work also uses a metric to decide processor allocation, but only one: MPI, which was found to work best. Allocation is based on mimicking the actions of an optimal scheduler to reduce performance degradation.

Klug et al. [38] implemented a scheduling system which forces pinning of tasks to specific CPU cores. After a time constant, their autopin system changes the pinning so various pinning combinations are tried. Once the best pinning combination is discovered, the tasks use it until they terminate. Experiments conducted with SPEC OMP benchmarks show substantial improvements over default Linux scheduler pinning behavior. The autopin approach will not be so useful if application behavior changes significantly after an ideal pinning combination has been discovered. Our multicore scheduler is better than the autopin approach because the decisions are made fast, without having to run tasks for long periods of time. The decisions are also more intelligent; autopin simply tries out different allocations to come up with the most suitable configuration.

The closest work to ours is that done by Blagodurov et al. [11, 15]. In their study, the authors investigated the best approaches to classifying threads, based on performance metrics obtained via CPU counters on modern Chip Multiprocessors (CMPs). These include metrics such as Cycles Per Instruction (CPI), Instructions Per Cycle (IPC), and Misses Per Instruction (MPI), used to estimate number of cache misses. The classification approaches are used to classify threads when they compete for resources. The classification approaches studied were:

- i. Stack Distance Competition (SDC). This method maintains cache models used to compute degradation of co-running tasks based on cache occupancy when they run alone.
- ii. Animal Classes. This method identifies four classes of tasks named *turtle*, *sheep*, *rabbit*, and *devil*, based on cache miss rates and how sensitive they are to cache allocation. Using a table, compatible animal classes are selected to run together.
- iii. Miss Rate. This method identifies tasks with high miss rates, since they tend to worsen the performance degradation due to memory controller contention, memory bus contention, and prefetching hardware contention. Therefore one high miss rate task and one low miss rate task can run together, but two high miss rate tasks should not run together.
- iv. Pain Classification Scheme. This method estimates the “pain” of co-scheduled tasks based on cache sensitivity (how sensitive a task is to cache space being taken away from it) and cache intensity (how much a task hurts another one by taking away its cache space). The goal when scheduling would be to use schedules with low pain.

Based on the classification study, the authors developed a scheduling algorithm known as Distributed Intensity (DI). Their results showed that its performance was close to the Integer Programming-based optimal solution. In our work, we investigate how task degradation could be predicted using machine learning approaches. We then selected the best model for use in a scheduler that combines DI with linear programming, and also to generate workloads to train Multilayer Perceptron and SVM schedulers, giving better results, approaching the optimal scheduler.

Kim et al. developed a method for the Multiprocessor Scheduling Problem, based on bat intelligence [39, 40]. They formulated a multi-objective optimization problem consisting of objectives to reduce makespan, tardiness, and energy consumption. They used an innovative optimization method called Bat Intelligence (BI) which mimics bat hunting behavior which uses echolocation to detect prey and CATD (Constant Angle Target Direction) to approach a randomly-moving prey efficiently with a fixed angle of attack. Analogous to echolocation, BI generates a designated number of feasible solutions. The best solution is chosen and from that, one element, a task and voltage scale level, is chosen to be part of a common element set. This element would be part of the solutions generated in subsequent iterations. This is analogous to CATD. When the common element is chosen, a termination condition is tested for and if met, the current cycle terminates. Overall termination occurs after a fixed number of cycles, or after a certain number of cycles with no progress. BI has been demonstrated as useful for multi-objective optimization, as well as for single-objectives, such as makespan, reducing the length of the job execution time. The disadvantage of BI is that it requires prior knowledge of execution rates of the various tasks on the processors. Our Processor Allocation approach only needs to be able to measure a single metric, MPI, and use that to assign tasks to cores, without prior knowledge of execution rates.

Jiang et al. [20] carried out a study on optimal scheduling of tasks on a multicore system. Their optimal solution is based on Integer Programming. The objective function and constraints, as well as a description of the problem formulation are given in Section 6.4.1.

2.5 Machine Learning in Scheduling

Rai et al. [41, 42] carried out studies on characterizing and predicting L2 cache behavior. They used machine learning algorithms to build models which could then be used for characterizing and predicting. The machine learning algorithms they used were Linear Regression, Artificial Neural Networks, Locally Weighted Linear Regression, Model Trees, and Support Vector Machines. The methods generate regression models, which involves fitting a model that relates a dependent variable y to some independent variables, x_1, x_2, \dots, x_n . The model is expressed in the form of an equation:

$$y = f(x_1, x_2, \dots, x_n) \quad (2.1)$$

The class variable to be predicted was named “solo run L2 cache stress”. The attributes (independent variables) used were

- i. L2 cache references per kilo instructions retired.
- ii. L2 cache lines brought in (due to miss and prefetch) per kilo instructions retired. This shows the stress put by a program on the L2 cache.
- iii. L2 cache lines brought in (due to miss and prefetch) per kilo L2 cache lines referenced. This shows the re-referencing tendency of a program.
- iv. Fractional L2 cache occupancy of a program. This gives a rough estimate of fraction of space occupied by a program in the L2 cache while sharing with another program.

Our multicore scheduling work focuses on predicting degradation of task performance and optimal scheduling decisions using the MPI metric.

Rai et al. [43] extended their work by developing a machine learning based meta-scheduler. They used their predictive models mentioned in the previous paragraph to aid scheduling decisions, achieving a 12% speedup over the Linux CFS scheduler. Their meta-scheduler divides tasks into two groups: those with high solo run L2 cache stress and low solo run L2 cache stress. This serves to reduce cache contention and competition for shared resources, thereby improving performance. Our multicore scheduler is focused on batch scheduling and instead of trying to beat the Linux scheduler, which is not a batch scheduler and not a contention-aware scheduler, tries to approach the optimal multicore scheduler at minimal cost.

Apon et al. [44] studied the use of a Stochastic Learning Automaton (SLA) for assigning tasks to parallel systems. The automaton keeps track of various possible states (for example, number of tasks executing, number of tasks waiting, etc.), possible actions (number of processors assigned to tasks), a matrix with probabilities of choosing actions from each state, responses on whether an action was “good”, “bad”, or “neutral”, and an updating scheme for updating probabilities in the matrix. Their results showed substantial improvement of the SLA approach, with varying probabilities, compared to a fixed assignment scheme. Our multicore scheduler can also be used for dynamic scheduling, but uses different approaches, based on machine learning models such as the Multilayer Perceptron, Support Vector Machine, and Model 5 Prime decision trees.

2.6 Summary

There has not been very much research done on tuning operating system schedulers. We have therefore also included research on tuning software systems in our literature review. Some available methods include on-line self-tuning, but these require operating system scheduler-specific implementations. The work in this thesis focuses more on a methodology for off-line tuning, which can easily be applied to a variety of operating system schedulers, provided they have tuning capabilities.

Chip Multiprocessors are the immediate future of CPU development, with more and more cores being incorporated into CPUs. With this development come some problems: how to manage CPU resource contention; in other words, how to design contention-aware schedulers. Current research has focused on theoretically optimal solutions, and fast on-line heuristics, which either try a variety of Processor Allocation configurations in a brute-force approach, or try to predict co-scheduled task degradation, based on performance metrics such as IPC and MPI. We try to combine these approaches by using machine learning models to come up with scheduling decisions close to the optimal scheduling decisions, but much faster.

Chapter 3

Experimental Approach

This chapter discusses our experimental approach towards meeting the various research objectives discussed in Chapter 1. Section 3.1 gives an overview of our approach. Section 3.2 gives an overview of the LinSched simulator and discusses two approaches we experimented with for modification of the LinSched simulator in order to accommodate the hackbench benchmark, thereby validating the simulator.

The LinSched validation section is organized as follows. In sub-section 3.2.3, we discuss some prior work on simulation and benchmarking of schedulers; sub-section 3.2.4 discusses process scheduling in the Linux operating system; sub-section 3.2.5 discusses the simulator we modified for this study, LinSched; sub-section 3.2.6 discusses our two approaches to implementing a scheduling benchmark in LinSched; sub-section 3.2.7 has our results, and the last part of the LinSched validation sub-section, 3.2.8, has our conclusion for the Section.

Section 3.3 discusses the AKULA simulator, in the context of our use of it. Section 3.4 gives an overview of the various benchmarks used in our experiments. Section 3.5 gives an overview of the hardware we ran our experiments on. Section 3.6 summarizes our experimental approach.

3.1 Approach Overview

In order to achieve our objective of developing a methodology for tuning an Operating System scheduler for a specific workload, we draw on the literature available for Operating System scheduler tuning as well as tuning of software and other systems, since,

to the best of our knowledge, very little work has been done in the area of Operating System scheduler tuning.

We select Response Surface Methodology (RSM) because this method can dramatically reduce the number of experiments required to be performed to find optimal values of system parameters [45]. Our proposed methodology is tested on a live system running the Linux Operating System, kernel version 2.6.28, which includes a scheduler, CFS (Completely Fair Scheduler), with several knobs used to tune it. A benchmark is selected based on popularity for use in evaluating Operating System schedulers. The selected benchmark is hackbench [46] and it has also been used by CFS developers for evaluating the scheduler [47, 48]. With RSM, a set of experiments is designed with various experimental values for the three scheduler knobs, in various combinations. A response, or performance value, for each experiment is obtained empirically. RSM uses optimization to obtain an optimal value for the various knobs. This new experiment is then compared with the benchmark evaluated running on the same system, but with default values for the knobs. Results show that the RSM-obtained knob values result in much better performance.

We again draw on the literature to develop a tuning methodology based on optimization methods. Methods selected are Particle Swarm Optimization, because it fits in well with the problem specification, where particles are used to “hunt” for ideal knob settings, and has good ability to avoid finding global minima as opposed to local minima [49], and Golden Section, because it offers even faster optimization, given that the operating system scheduler tuning problem can be converted from a three-dimensional optimization problem, to a one-dimensional optimization problem. This is possible because our review of the Linux operating system scheduling literature and Linux kernel source code showed certain relationships between the knobs. This time a live system was not used, but rather a scheduling simulator. This was done to speed up experiments, which could run for many hours, and for convenience of experiments; other tasks can run on the machine at the same time. The simulator used was LinSched, which was developed by extracting the CFS scheduler from the Linux 2.6.23 kernel and converting it into a stand-alone user-mode process. We ported the hackbench benchmark so it runs within LinSched. Again, the hackbench benchmark was chosen because it has been used to evaluate the performance of the CFS scheduler by scheduler developers.

3.2 The LinSched Simulator: Comparison of Bootstrapping and Finite State Machine Simulations of a Scheduling Benchmark

We draw on the literature to study the Processor Allocation problem, to discover the current state-of-the-art and various issues around the problem. Machine Learning has not been used extensively in this area; rather various other heuristics are used. We experiment with various Machine Learning algorithms. Our experimentation environment for this part of our research is the AKULA scheduling simulator. This simulator was chosen because it implements a state-of-the-art scheduling approach called bootstrapping, which accurately estimates task execution progress when co-scheduled with other tasks on cores of a CMP which share certain critical resources, such as the Last Level Cache (LLC). Benchmarks chosen to evaluate our scheduling approaches come from the SPEC CPU 2006 benchmark suite. We train several Machine Learning models including the Multilayer Perceptron (MLP) and Support Vector Machine (SVM), which are then used to generate scheduling decisions. Inputs into these models include certain task attributes: MPI (Misses Per Instruction). For training purposes, we generated artificial data using another Machine Learning algorithm: M5P (Model 5 Prime). We therefore had up to 80,000 instances in the training data set. Training and evaluation was done using 5-fold Cross Validation. AKULA was modified a great deal to accommodate our experiments. We attempted to prototype a scheduler which approached an Optimal Scheduler (which was discovered by studying the literature) in performance of generated schedules, thereby also outperforming a state-of-the-art contention-aware scheduler called Distributed Intensity (DI) reported in the literature. We achieved this. For a comparison we implemented the Optimal Scheduler and the DI scheduler.

3.2 The LinSched Simulator: Comparison of Bootstrapping and Finite State Machine Simulations of a Scheduling Benchmark

3.2.1 Overview

The LinSched simulator was selected for evaluating the standard optimization-based scheduler tuning approaches, as explained in Section 3.1. This sub-section, 3.2, compares simulation of a scheduling benchmark on the LinSched simulator using two different approaches. The benchmark mimics the behavior of a chat application, with

3.2 The LinSched Simulator: Comparison of Bootstrapping and Finite State Machine Simulations of a Scheduling Benchmark

sender and receiver threads. One approach involves representing the benchmark as a Finite State Machine (FSM) [50], in which the various threads could have different states, and move from one state to another until they complete. Another approach uses a benchmark performance approximation method known as bootstrapping, introduced in the AKULA scheduling simulator. The benchmarks are simulated in the LinSched Linux scheduler simulator. Our results show that bootstrapping produces results (actual runtime figures) closer to those obtained when running the actual benchmark on an operating system. The simulation also completes much faster with bootstrapping. However, the Finite State Machine results in a better model of scheduling, with higher significance and better R and R^2 values, and a better regression fit which is easily observed.

3.2.2 Need for LinSched and Simulated Benchmarks

The operating system is a very important part of the computer system. Its job is to manage the resources, such as the storage devices, the main memory, and the CPU [2, 3, 4]. The component of the operating system responsible for managing the CPU is the process scheduler, which we will refer to as the scheduler from now on. There are many different scheduling algorithms in existence. Each algorithm works well for certain workloads, while others work better for other workloads. Scheduling algorithms are judged on their suitability for handling workloads based on various performance metrics, such as throughput (the number of processes that complete per unit time), turnaround time (the time it takes for a particular process to complete, from start to finish), CPU utilization (fraction of time the CPU is doing useful work), and response time (the time it takes for a process to respond to some event, from the time the event occurs, to the time the response is produced) [2, 3, 4].

Developing and testing schedulers is time consuming and involves modification of operating system kernel source code, followed by compilation, and then rebooting and restarting of the operating system [51]. Bugs present could be hard to detect and could bring down the system, since the scheduler code runs in the kernel and has high privileges, enabling it to affect other components. Debugging the operating system is also difficult. Because of these reasons, simulators are used, in order to isolate the scheduler. These simulators enable convenient development and testing of schedulers. However, whether using a simulator or not, the scheduler must be supplied with a

3.2 The LinSched Simulator: Comparison of Bootstrapping and Finite State Machine Simulations of a Scheduling Benchmark

workload, i.e. a set of processes, which will execute, and by which one can judge the schedulers performance. Such workloads are called benchmarks. In operating system scheduler simulation, the benchmarks can also be simulated. In this section we compare two approaches to simulating a well known scheduling benchmark. One approach uses a Finite State Machine (FSM) implementation of the benchmarks, while the other approach uses implementation based on bootstrapping, a simulation approach introduced to the scheduling community in 2010.

3.2.3 Related Work

AKULA is a scheduler evaluation tool developed by Zhuravlev et al. [12]. It is Java-based and offers easy coding of algorithms together with rapid evaluation using a novel method known as bootstrapping, described in more detail in Section 3.2.6.2. AKULA can also execute a scheduler on a live platform by tying process tasks to cores on multicore CPUs and evaluating their performance. AKULA is mainly useful for evaluation of schedulers for multicore architectures. We chose to use LinSched as our simulator for scheduler tuning because it already incorporates the Linux scheduler code, which would take time to port to AKULA. Magnusson et al. [52] describe Simics, a full system simulator. This provides for a detailed simulation of an entire system, complete with a base hardware layer. Such simulators are capable of executing instructions and can therefore run actual binary images of programs; no need to simulate benchmarks. However, they are slow, and require fast platforms to run. They can also be difficult to work with. They also model many details not required to evaluate operating system schedulers [12]. Our use of LinSched avoids these disadvantages. CMPSched\$im is a tool developed by Intel for studying scheduling and interaction with caches [53]. It combines a binary instrumentation tool, Pin [54], LinSched, and a cache simulator. It is capable of executing benchmark binaries and evaluating performance, which is dependent on how the tasks are scheduled. It is a very powerful tool, but unfortunately Intel chose not to make it available to the research community at this stage. Hence, we have to develop our own tools or modify existing tools.

3.2.4 Process Scheduling in the Linux Operating System

Starting from Linux kernel version 2.6.23, a scheduler known as the Completely Fair Scheduler (CFS) has been used as the default Linux scheduler [55, 56]. The CFS tries to

3.2 The LinSched Simulator: Comparison of Bootstrapping and Finite State Machine Simulations of a Scheduling Benchmark

ensure all tasks have a fair share of the CPU i.e. process A should not be kept waiting for a long time while process B keeps getting opportunities to use the CPU. The CFS scheduler maintains a red-black tree (a binary search tree) that orders tasks according to how unfairly they are being treated. A task that is being treated unfairly because it has been kept waiting for a long time will move to the “front” of the red-black tree, and will soon be selected to execute on the CPU. The CFS scheduler is easily extensible.

3.2.5 The LinSched Simulator

LinSched is a Linux scheduling simulator developed by Calandrino et al. [51], and is based on the Linux 2.6.23 CFS scheduler. The CFS scheduler has been extracted from the Linux kernel source code, including all the C header and source files and kernel configuration files. Additional simulation code has been written to handle the execution of tasks. An API (Application Programming Interface) has been created to enable users to create their own tasks and run the simulation for a specified period of time. When a task is created, the scheduler is made aware of this, since it needs to schedule it. Once the simulation has started, the simulator advances the virtual time and for each advance, the scheduler is invoked to select and execute tasks for each CPU in the system.

LinSched has many advantages for use as a research tool over a live operating system. The scheduler can be modified and tested easily. LinSched runs as a process in user space, just like any other application. A bug in the scheduler's code only affects this process. In the worst case only this process crashes and can easily be restarted. The cycle of code, compile, reboot, benchmark, debug, commonly used for live operating system kernels is not present, since there is no need to reboot. Debugging is also easier. LinSched allows fast validation of the scheduler, since the scheduler is isolated. Other simulators exist, such as AKULA [12], but require the CFS scheduling algorithm to be ported to them. With LinSched, the CFS algorithm is already part of the software. There are currently projects to update LinSched to incorporate more recent CFS versions.

3.2.6 Benchmark Simulation

In order to evaluate the performance of a system, a workload has to be used to stress the system. Such a workload should be similar to live workloads that will run on the

3.2 The LinSched Simulator: Comparison of Bootstrapping and Finite State Machine Simulations of a Scheduling Benchmark

system when it is in operational use and is called a benchmark [25]. When using a simulator for performance evaluation, the benchmark is usually simulated i.e. it does not involve use of live processes with instructions, which the CPU will execute. Rather, it uses a trace of events obtained when the benchmark was running on a live system, or characteristics of tasks, such as the points in time when they are created, and the amount of CPU time each of them requires in order to do its work. The benchmark we selected for our study is hackbench [46]. It has been used extensively to benchmark the CFS scheduler by the creator of the CFS scheduler, Molnar [57, 47, 48]. Hackbench mimics the behavior of a chat server. By default it has five groups of 20 senders and 20 receivers each. Each sender sends a fixed number of messages to each receiver in its group, all 20 of them. Each receiver receives messages from all the senders in its group, all 20 of them. When the sending and receiving is done, the senders and receivers exit and the benchmark completes. Porting hackbench to LinSched presented a challenge since LinSched by default only handles simple tasks that do not have much logic in them. This study involved the following modifications to LinSched:

- i. Implementation of a sender task callback and a receiver task callback, which the scheduler invokes when these tasks are selected to use the CPU. We used two approaches: Finite State Machine (FSM) models and bootstrapping. For FSM, the sender and receiver task callbacks implemented the finite state models. For bootstrapping, a single callback function was used for both senders and receivers, which kept track of the progress. FSM and bootstrapping are explained in the next two sections.
- ii. Implementation of some helper functions e.g. to convert strings to integers, which was necessary because the C library for this function could not be linked because of conflicts with the kernel code.
- iii. Functions to check when all tasks have finished. These are lacking in LinSched, since benchmark tasks that run to completion were not catered for.
- iv. Data structures to keep track of various task states and groupings in FSM.
- v. Data structures to keep track of progress and groupings in bootstrapping.

The following two sections describe our approaches to implementing hackbench in LinSched.

3.2 The LinSched Simulator: Comparison of Bootstrapping and Finite State Machine Simulations of a Scheduling Benchmark

3.2.6.1 Finite State Machine Approach

A Finite State Machine, also known as a finite automaton, is a model of a system, where the system is described as being in one of several states [50]. There is a start state, and then for each input, the system either remains in the same state, or moves to another state. Eventually after one or more such transitions, the system exits. This model maps very well to the way the sender and receiver threads in the hackbench benchmark work. Fig. 3.1 shows the FSM (Finite State Machine) model for the sender and Fig. 3.2 shows that for the receiver.

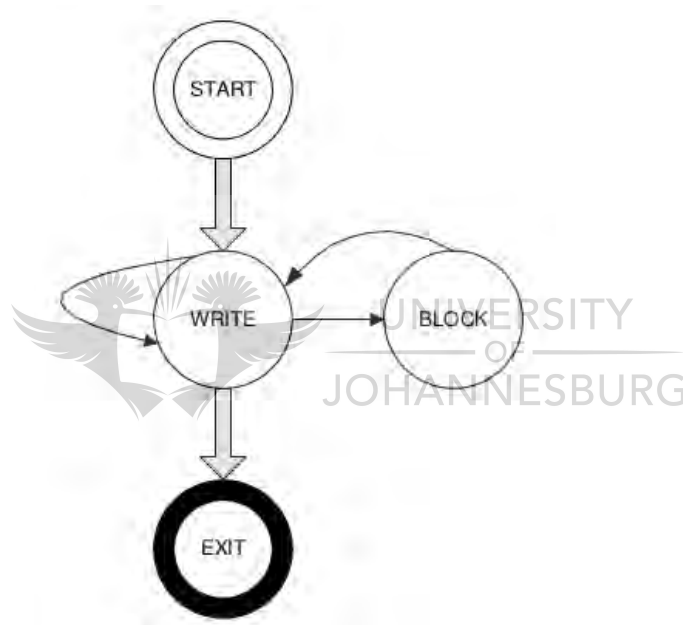


Figure 3.1: Finite State Machine for Sender

The sender is created in the START state, and then it moves to the WRITE state, in which it tries to WRITE to a receiver. If the receiver is not ready e.g. it is busy receiving from another sender, the sender moves into the BLOCK state where it waits. If the receiver accepts the data, the sender remains in the WRITE state. If the sender is blocked and the receiver it was trying to write to becomes available i.e. it is no longer busy, the sender again moves to the WRITE state to continue writing. After the sender has written (sent) all its messages to all the receivers in its group, it exits i.e. enter the EXIT state. The receiver also starts in a START state. It then moves to a BLOCK

3.2 The LinSched Simulator: Comparison of Bootstrapping and Finite State Machine Simulations of a Scheduling Benchmark

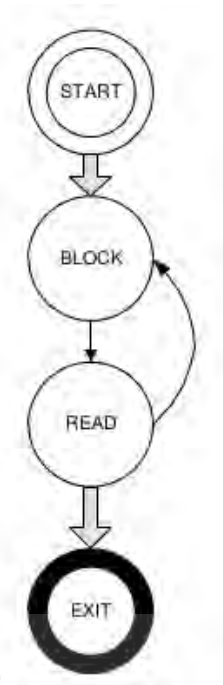


Figure 3.2: Finite State Machine for Receiver

state to receive data. When the data arrives it reads it in the READ state, and then BLOCKs again for the next piece of data. Once all data has been read, it EXITS.

3.2.6.2 Bootstrapping

Bootstrapping is an approach for simulation of workloads running on an operating system with a specific scheduling algorithm [12]. It is aimed at analyzing scheduling algorithm behavior on multicore systems. Such systems consist of one or more memory domains. Each memory domain could contain one or more cores. Each core is capable of executing instructions belonging to a thread. The cores in a memory domain typically share some hardware resources such as a cache, memory controller, and pre-fetch hardware. Since these components are shared, when two or more threads run in the same memory domain, they experience a negative performance impact. Bootstrapping keeps track of progress of various threads by calculating the impact on progress when several threads are scheduled in the same memory domain. The bootstrapping method involves profiling, where a benchmark thread is run first off all on its own in a memory domain, followed by running it together with another benchmark thread. Then the

3.2 The LinSched Simulator: Comparison of Bootstrapping and Finite State Machine Simulations of a Scheduling Benchmark

degradation is measured. For tasks X and Y , the formula is given in [12] and is shown below.

$$Deg(X, Y) = \frac{RuntimeAlone(X)}{RuntimeCombined(X, Y)} \quad (3.1)$$

The simulator keeps track of time using ticks. A tick is an interval of time between when the hardware timer gives control back to the operating system via an interrupt. At every tick, the scheduler is brought into play to decide whether to take the CPU away from a task, or to calculate the progress of all the tasks. The formula used to calculate the progress is given in [12] and is as follows:

$$P(X) = 100\% \times \frac{tick}{Solo(X)} \times Deg(X, N(X)) \quad (3.2)$$

$P(X)$ is the progress of task X , $tick$ is the length of the tick interval, $Solo(X)$ is the runtime of X when running alone in a memory domain, $Deg(X, N(X))$ is the Degradation when X runs together with the Neighbour of X ($N(X)$). Bootstrapping, as a method for calculating progress of simulated tasks based on co-run degradation, was introduced first in 2010 by Zhuravlev et al. [12]. In order for our implementation to work, we measured the performance of hackbench when one group runs alone. We then measured the performance when one group is scheduled with another group i.e. two groups of senders and receivers run at the same time. We calculated the degradation. In bootstrapping, one has to measure the degradation for all possibilities of benchmarks running with other benchmarks. In our case, since we were testing on a dual-core machine i.e. it has two cores in the same memory domain, and all the groups are from the same benchmark, we only had to take those two measurements: one group solo, and two groups together.

3.2.7 Results

We conducted our experiments on a machine with Intel Core 2 Duo CPU T8300 (2.4GHz and 3MB L2 Cache) and 4GB of RAM. The operating system was Ubuntu Linux 9.04. We used a configuration of hackbench where each sender sends a 1,000 messages to each receiver. Fig. 3.3 shows a plot of actual hackbench execution runtimes for 2, 4, 6, 8, ..., 20 hackbench groups versus the LinSched runtimes, after modification for FSM. LinSched shows runtime results in Jiffies, which depend on the frequency of the

3.2 The LinSched Simulator: Comparison of Bootstrapping and Finite State Machine Simulations of a Scheduling Benchmark

hardware clock. For our implementation, 1 Jiffy = 1 millisecond. A linear regression model was produced with high significance for the intercept. The model produced for FSM was significant at the $p < 0.001$ level, showing a good fit. R^2 and Adjusted R^2 were close to 1 (both 0.9997), also indicating a good model. Fig. 3.4 shows a plot of actual hackbench execution runtimes for 2, 4, 6, 8, ..., 20 hackbench groups versus the LinSched runtimes, after modification for bootstrapping.

The model produced for bootstrapping was also significant at the $p < 0.001$ level, showing a good fit. R^2 and Adjusted R^2 were close to 1 (0.9881 and 0.9866 respectively), also indicating a good model. As can be seen in Fig. 3.3 and 3.4, the FSM model is a better fit than the bootstrapping model. Note that the Jiffies displayed by LinSched differ in magnitude from live hackbench runtimes because of the operation of the simulator. Hence we need to fit a model in order to be able to translate from Jiffies to actual milliseconds.

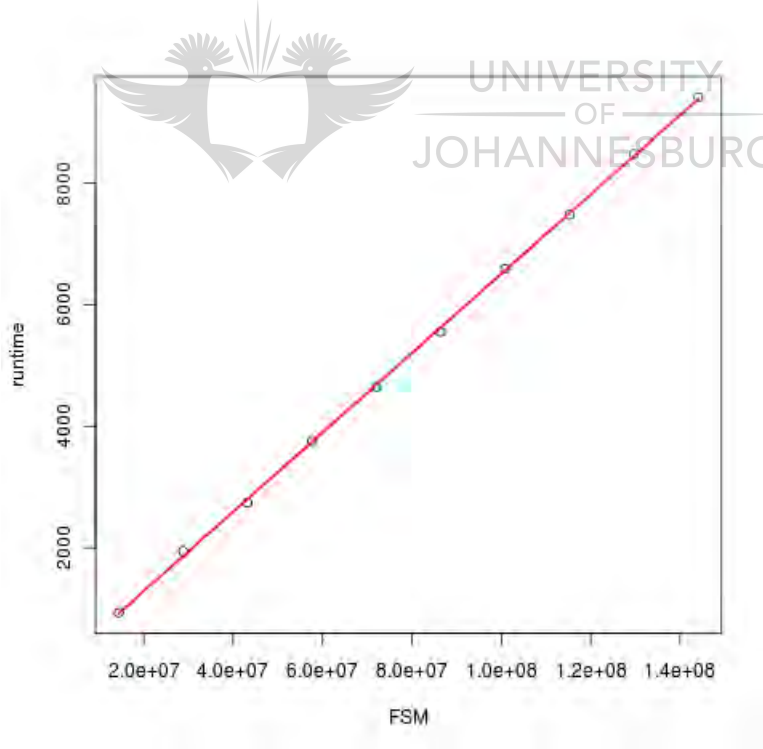


Figure 3.3: Regression Model for FSM. Runtime in milliseconds vs. FSM Jiffies

Our degradation (see Section 3.2.6.2) was 0.528, meaning that when two hackbench

3.2 The LinSched Simulator: Comparison of Bootstrapping and Finite State Machine Simulations of a Scheduling Benchmark

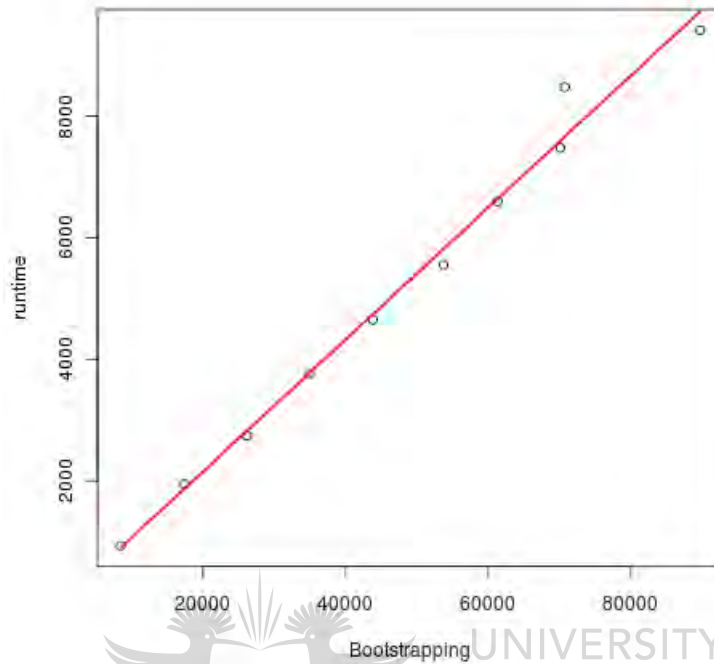


Figure 3.4: Regression Model for Bootstrapping. Runtime in Milliseconds vs. Bootstrapping Jiffies

groups run together, they will finish in just under twice the time for one group. With the largest load (20 groups) FSM LinSched takes 53 seconds to execute, while Bootstrapping LinSched takes 3 seconds. Our results show the following:

- i. FSM LinSched produces more reliable results than bootstrapping.
- ii. Bootstrapping LinSched executes much faster.
- iii. The bootstrapping implementation is shorter than FSM (149 vs. 105 lines of code), making coding and debugging easier. Lines of Code is an important metric used to judge productivity of Software Engineers [58].

3.2.8 Conclusion

In comparison, a bootstrapping implementation of the hackbench benchmark on LinSched produces results (actual runtime figures) closer to those obtained when running

3.3 AKULA Multicore Scheduling Simulator

the actual benchmark on an operating system. The simulation also completes much faster with bootstrapping (3 seconds versus 53 seconds with FSM LinSched, with our benchmark). However, the Finite State Machine results in a better model of scheduling performance, with higher significance. Therefore, if speed is of the essence, bootstrapping can be used. Its creators, Zhuravlev et al. [12], emphasize that it gives rough approximations of performance, useful for comparing scheduling algorithms together. If accuracy is paramount, the FSM approach is better. The bootstrapping implementation is much shorter than the FSM one, with less code and therefore less opportunity for bugs to creep in.

3.3 AKULA Multicore Scheduling Simulator

3.3.1 Overview

For developing solutions to the core-assignment (Process Allocation) problem, we made use of the AKULA [11, 12, 15, 19] scheduling simulator.

The AKULA simulator was selected because it uses a very powerful approach to multicore scheduling simulation called *bootstrapping* not found in other simulators. The AKULA bootstrapping feature is described in Section 3.2.6.2.

3.3.2 Validation

The AKULA simulator was validated by its authors, as reported in [12, 19]. They gathered data on run times of various benchmarks when run on their own, and when co-scheduled with other benchmarks. For this they used the AKULA profiling functionality, which precedes bootstrapping in the simulation process. They then simulated the benchmarks using bootstrapping, with different scheduling algorithms. After this, they ran the same benchmarks in the same combinations (solo and then co-scheduled) on an actual system, using the same set of scheduling algorithms. They compared results obtained from bootstrapping and actual execution and found that the AKULA simulator can be reliably used to evaluate scheduling algorithms.

3.3.3 AKULA Framework

The AKULA simulator is implemented in the Java programming language. It has an object oriented design consisting of 15 classes. Table 3.1 lists the classes and shows

3.4 Benchmarks

modifications made.

Table 3.1: AKULA Classes and Modifications Made for this Thesis

#	Class Name	Function	Modification Made
1	AKULA_Thread	Stores Details of Threads	Attribute to store MPI of task added
2	bootstrap	Uses bootstrap data to run and evaluate a scheduler	Modified to run each task at least 3 times, so all finish at the same time
3	bootstrap_DB	Instance of bootstrap database that keeps track of one metric	None
4	chip	Abstracts memory domains, contains cores	None
5	core	Abstracts individual cores, part of machines	None
6	daemon	Works with daemon that attaches performance counters	None
7	helper	Some helper methods for assertion	None
8	history_node	used to keep track of how threads were scheduled	None
9	home	main class	Modified to read multiple workload configurations, read saved machine learning models, and invoke multiple simulations serially
10	initializer	reads in data about performance of threads when running solo and co-scheduled	Modified to use machine learning models to predict degradation
11	machine	Top level object for memory domains	None
12	profiler	Gathers performance data about running threads	None
13	scheduler	Implements a scheduler	Modified to implement various scheduling algorithms using Linear and Integer Programming, MLP, and SVM
14	stats	Calculates statistics about simulation	Modified output format
15	wrapper	Used to run scheduler on real machine	None

3.4 Benchmarks

We used benchmarks from the SPEC CPU 2006 benchmark suite [59]. Nine benchmarks were selected to give a balance between memory intensive and non-memory intensive applications [11]. These benchmarks are *gcc*, *lbm*, *mcf*, *milc*, *povray*, *gamess*, *namd*, *gobmk*, *libquantum*. Out of these, the four most memory intensive applications are *lbm*, *mcf*, *milc*, and *libquantum*. Therefore when these are scheduled to run on the same chip (sharing a cache), there is severe performance degradation.

These benchmarks were supported in the original AKULA release [60]. Various attributes such as MPI (Misses Per Instruction) and IPC (Instructions Per Cycle), MPC (Misses Per Cycle) and solo runtime length were made available in the release.

3.5 Experimental Platforms

Experiments with the LinSched simulator were conducted on an Intel Core 2 Duo T8300 CPU laptop (2 cores, 2.4GHz, 3MB L2 Cache, and 4GB RAM), running the Linux Operating System, for the RSM and standard optimization approaches.

Experiments with the AKULA simulator were conducted on an Intel Core 2 Duo T8300 CPU laptop (2 cores, 2.4 GHz, 3MB L2 Cache, and 4GB RAM) and an Intel Core i7-2600 CPU desktop (4 cores, 3.4 GHz, 4MB L2 Cache, and 16GB RAM), running the Linux Operating System.


3.6 Summary

We adopted an experimental approach to studying the Operating System scheduling problems identified in Chapter 1. This involves selecting of tuning and Processor Allocation methods, and evaluating them. The methods are implemented in the LinSched scheduling simulator, for RSM and standard optimization-based tuning, and the AKULA simulator for Processor Allocation. In order to obtain performance results, standard benchmarks are used, namely hackbench and nine benchmarks from the SPEC CPU2006 benchmark suite. The simulators used had to be modified substantially in order to accommodate our study.

Chapter 4

Operating System Scheduler Tuning Using Response Surface Methodology

4.1 Summary



Tuning operating system components is a cyclical process involving setting parameters, evaluating the effect of the settings, making adjustments, and testing again. This is an expensive process, both taking a long time and requiring money to hire people to do it. In this chapter we present a statistical approach to tuning of an operating system scheduler using Design of Experiments (DOE) and Response Surface Methodology (RSM). We make use of a benchmark and generate a response surface based on the runtime of the benchmark and three Linux scheduler parameters. We produce a model of the scheduler and optimize the parameter settings, minimizing the number of times the benchmark had to be run to find the optimal settings. In our experiment, we achieved a **16.48%** performance improvement when the Linux scheduler runs the benchmark. We also compared the scalability of the optimized and unoptimized schedulers and discovered that the optimized scheduler does much better in this regard. This was done without prior knowledge of optimal settings for the workload we used.

This chapter is organized as follows: Section 4.2 gives an overview of the problem being solved. Section 4.3 gives an overview of Response Surface Methodology. Section 4.4 has a literature review on use of Response Surface Methodology and other

statistical methods in software tuning. Section 4.5 goes over the functionality of the Linux scheduler and various options for tuning its operation. Section 4.6 discusses the design of the experiment used to capture the effect of various tuning parameters on performance. Section 4.7 discusses the analysis of the tuning model, which validates its usefulness statistically. Section 4.8 discusses the benchmark environment and the results of using the model to optimize the performance of a system running a workload on top of the Linux operating system. Section 4.10 has our conclusion.

4.2 Scheduler Tuning Introduction

Operating systems consist of several components such as the scheduler, memory manager, file system manager, and I/O device manager. These components have several parameters which are used for tuning purposes to suit a particular workload. A workload consists of units of resource utilization, such as processes and threads, which will make use of the various resources in the computer through the appropriate resource managers (scheduler, memory manager, etc.) [2, 3, 4].

In this chapter, we consider one of these resource managers: the CPU scheduler. Schedulers, for example the one in the Linux operating system kernel, have various parameters that are set with no efficient formal methodology governing these settings; operating system performance parameters, including scheduler parameters, are set in an iterative cycle of tuning and measuring [6, 7]. This involves measuring components of the system such as CPU utilization, memory usage, and throughput. This iterative process consumes a lot of time, involving repetitive phases of measuring and tuning [8, 23, 24]. It would be of great benefit to system engineers and administrators to have a tool that could help to tune parts of an operating system optimally. In this chapter we focus on tuning the operating system scheduler, using the Linux scheduler to demonstrate a proof of concept.

We are proposing a tool based on Response Surface Methodology (RSM) that could be used to tune an operating system scheduler to give optimum performance for certain workloads. RSM is ideal for the purpose of obtaining system parameter values using a limited and small number of experiments, and has been used for such purposes in the literature [26, 27, 28]. RSM has not been used for Operating system scheduler tuning. Such a tool will help to back up parameter-setting with a sound foundation,

based on a statistical approach, which will generate a set of parameter settings to test the scheduler with, and then decide on the parameter settings that give an optimal response. The response we are concerned with is the turnaround time of a workload i.e. how fast the workload completes execution.

4.3 Response Surface Methodology Overview

Response Surface Methodology follows from Design of Experiments (DOE), therefore DOE is explained first.

4.3.1 Design of Experiments

DOE is an approach to planning experiments in order to discover cause and effect relationships [61]. It can be used to investigate any process with measurable inputs and outputs. Given an experiment with 3 factors, A, B, and C, an experiment can be designed with each of the factors varied over two levels, high and low. There will be 8 different runs to accommodate this. Such a design provides contrasts of averages, thereby giving it statistical power. The alternative is to use One-Factor-At-a-Time (OFAT) experiments, where one of the factors is varied while the others are maintained at a certain level. One problem with this approach is that it requires more runs to get the same statistical power as DOE. The additional runs come in the form of replications of experiments, needed to come up with averages. Therefore, with 3 factors, DOE would need 8 runs, while OFAT would need 16. Another major problem with OFAT is that it cannot capture effects of interactions of factors. For example, executing a run with either A or B at a high level, while keeping the other factors at low levels may be misleading; the result of the experiment may be different if both A and B (i.e. AB) are at high levels at the same time. OFAT does not capture this. The results of DOE can be used to discover factors that have significant effects on the outcome of experiments, as well as their appropriate levels. They can be used to create a first order model, one in which variables are raised to the first power, which is 1.

The following steps are followed for two-level factorial designs (these are the types of experiments described above):

4.3 Response Surface Methodology Overview

- i. The effects are calculated. Averages of highs versus averages of lows, using the following formula [61]

$$\text{Effect} = \frac{\Sigma Y_+}{n_+} - \frac{\Sigma Y_-}{n_-} \quad (4.1)$$

where $\frac{\Sigma Y_+}{n_+}$ calculates the average value for an effect at a high level, which is averaged over the n factors involved. The Y s are the associated responses.

- ii. With the help of a probability plot, significant effects are identified. Effects are of the various factors on their own and their combinations.
- iii. The SS (Sum of Squares) for each effect is given in [61] and calculated as:

$$SS = \frac{N}{n}(\text{Effect}^2) \quad (4.2)$$

where N is the number of runs and n the number of data points collected at a level of the effect. The formula for an effect is given in equation 4.1 above.

- iv. Compute SS_{Model} , which is done by adding the effects for the significant effects.
- v. Compute $SS_{Residuals}$, which is done by adding the effects for the other effects.
- vi. Means Squares (MS) are also calculated (SS/df), making use of the degrees of freedom (df , the number of values available to calculate the residuals).
- vii. The ratio of Mean Squares ($MS_{Model}/MS_{Residual}$) provides the F-value for ANOVA.
- viii. With this information, an ANOVA table is constructed. This table provides the p-values, which show how significant each effect is in the model. The p-value gives the probability that the various metrics of a model are due to noise. So a p-value of 0.05, for example, means that there is 95% confidence in the model, and a 5% chance the results could be due to noise [61].
- ix. The results are then interpreted.
- x. The information from DOE can be used to produce a linear model useful for predicting results (responses) for given factor values.

However, when two-levels of factors are used, non-linear models cannot be produced. Three levels of factors can be used, but with many runs. This is where RSM comes into the picture.

4.3.2 Response Surface Methodology

RSM goes a step further than DOE, by producing a *response surface*. The response is some output, such as process yield. This is useful when the response (output) is not related to the factors in a linear manner. The response surface typically has some curvature. Quite often a quadratic equation will suffice, but other orders are possible, such as cubic and quartic [61]. A DOE design, as described in Section 4.3.1, cannot be used to fit surfaces defined by equations of second order or higher, such as quadratic equations, since it only has two levels for each of the factors (low and high). RSM requires at least three levels for each of the factors, which are: low, center, and high.

The following steps are followed for RSM [62]:

- i. Design an experiment and carry it out in order to generate response data, for each combination of factor levels.
- ii. Fit the data to a set of polynomials using regression tools.
- iii. Using ANOVA, compare the models to assess statistical significance and evaluate the lack of fit.
- iv. Choose the simplest model which predicts the response the best. Simple models are easier to interpret, easier to work with, and could generalize better.
- v. Carry out optimization, for example using the Simplex method [62].

Various designs are possible for RSM experiments [45, 62]. One is the Central Composite Design (CCD), shown in Figure 4.1. This design is built from a two-level factorial design (where each factor has two levels), plus center points, and axial points represented by stars, plus more centre points. In the Figure, each of the three factors involved is represented by an axis. The circles show possible values for each factor (low and high). In addition, there are centre points. The star points are values beyond the high and low bounds and improve the estimation of curvature. Another type of design is the Box-Behnken Design (BBD), shown in Figure 4.2. This requires only three levels for each factor. For our experiment we chose the CCD because of better estimation of curvature.

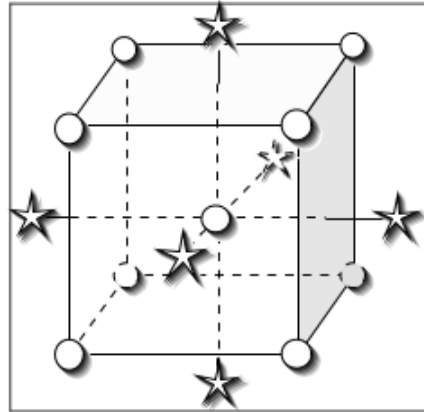


Figure 4.1: Central Composite Design (CCD) for three factors, with stars on axes. Circles and stars show possible settings for the factors.

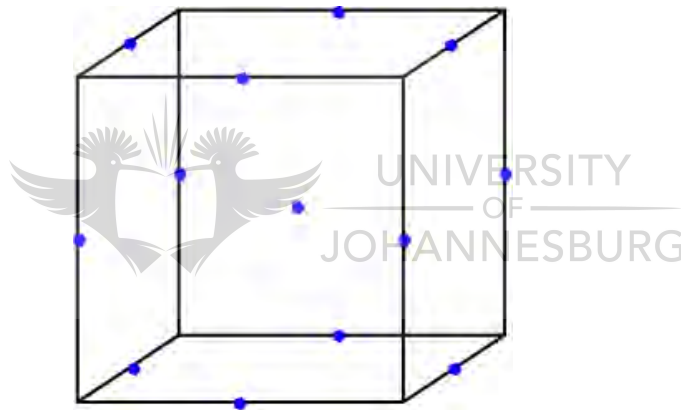


Figure 4.2: Box-Behnken Design (BBD) for three factors. Circles show possible values for the factors.

4.4 Use of Response Surface Methodology and Other Statistical Methods in Software Tuning

In this section we give a background on use of Response Surface Methodology (RSM) in the area of Computer Systems. This includes system software such as operating systems, database systems, networks, programming systems, etc. This is a shorter review than Section 2.2, with key points highlighted. Section 2.2 has more details.

Scientists used the OFAT (“One-Factor-At-a-Time”) approach for years to develop empirical models linking factors with responses [26]. The problem with this approach is

4.4 Use of Response Surface Methodology and Other Statistical Methods in Software Tuning

that it does not take into account the impact of interactions between factors on the response. A factor may not affect the response much on its own, but when combined with another factor, the impact could be significant. In this thesis OFAT is not followed, as that would require too many experiments and would take a very long time to complete. OFAT cannot therefore be prescribed as an approach to be used for operating system scheduler tuning.

Response Surface Methodology has been used in various studies for optimization tasks, though not for the operating system scheduler. Totaro [26] used RSM to configure multi-hop wireless mesh networks.

Vadde [27] presented a methodology to optimize the performance of the layered network protocol stack. Vadde used Design of Experiments (DOE) to learn about the various factors and find out which ones are more significant, followed by RSM to optimize the settings for factors in different layers. Our study is similar in that it provides a tool for performance optimization based on RSM, but it is different in that we are focusing on the operating system scheduler, not networks.

Shivam [28] developed a framework for managing benchmarking of systems based on RSM. They used an NFS (Network File System) server to demonstrate the usefulness of their approach. They cite the usefulness of RSM when applied to measuring system performance dependent on various configurations and workloads, listing its ability to “evaluate design and cost trade-offs, explore the interactions of workloads and system choices, and identify optima, crossover points, break-even points, or the bounds of the effective operating range for particular design choices and configurations” as big advantages. They tested their framework by seeing how fast it can find the saturation point for the NFS server i.e. the point at which the response time exceeds a threshold, meaning that this point represents the maximum load the server can handle. RSM is used to select combinations of levels of different factors (workload, physical resources and software configuration) to speed up the convergence to the saturation point.

Sullivan et al. [29, 30] developed a system for tuning software systems using influence diagrams. These diagrams are directed acyclic graphs that capture decision making, performance random variables, and decision maker’s utility. Decision making represents the knobs that are manipulated to tune the software system.

4.5 The Linux Scheduler

Linux kernel versions are given as three-number codes, separated by dots (x.y.z). For example, 2.6.23. The numbers further to the left represent more significant weights, so if the left-most number changes from 2 to 3, this has more significance than if the right-most number changes from 28 to 29. More significance means more new features. Our study uses the 2.6.28 kernel that comes with the Ubuntu 9.04 Linux distribution.

The Linux scheduler, over the years, has tried to maintain a balance between throughput and response time, thus trying to satisfy both desktop users and servers [63]. Response time is the time from when a user submits a request to the time the first response is produced [3, 2, 4]. It is a very important metric in interactive environments, such as desktop environments, where users use interactive applications, giving various commands and expecting fast responses. On servers, however, quite often throughput is considered more important, for example, on database servers or scientific computation servers used for weather modelling [25]. This metric gives the number of processes completed per unit time. In these environments, the faster the workload completes the better. With interactive workloads, after a user issues a command, the application does not exit. Rather it responds to the command, and continues running, waiting for more commands.

It is the job of the operating system scheduler to decide which process or thread should use the CPU next and for how long. To guide its decisions, it keeps track of various values, such as the priority of processes (how important a process is), the time assigned to a process to use the CPU and how long it actually used it for before giving it up, and how long a process has been kept waiting for the CPU [2, 3, 4].

The default Linux scheduler before the current one was called the $O(1)$ scheduler. This is because it performed a critical part of its operation, changing the status of all processes in the system, from a state where they have used up their time slices and are thus runnable (cannot compete for the CPU), to a state where they can compete for the CPU, in constant time, independent of the number of processes. The scheduler before the $O(1)$ scheduler, every once in a while, would keep processes waiting for the CPU while it calculated priorities and time slices, making the system non-deterministic, affecting the operation of the processes, especially the real-time processes [2]. The $O(1)$ scheduler was introduced in the Linux kernel in the 2.6.0 version. The current default

Linux scheduler is called the Completely Fair Scheduler (CFS). It was introduced in the 2.6.23 kernel version. CFS models an ideal multi-tasking CPU [57]. Such a CPU does not actually exist, because there are various overheads involved in running concurrent tasks; total runtime is not simply a sum of the various task execution times, but includes overheads such as context switch time spent saving and loading task states. The ideal CPU can run n processes, each at $1/n$ speed, in parallel. If there is one task running, it runs at 100% CPU power and finishes quickly. If two tasks are running, each runs at $1/2 = 50\%$ power, with both tasks using the CPU at the same time in parallel. In this model, since in an actual system with only one CPU only one task can run at a time, the scheduler keeps track of how long each process has been waiting. This time is the time the process should run in order for the system to be fair. CFS tries to run the task that has been treated unfairly the most, that is, the one with the largest waiting time value. The CFS scheduler maintains a red-black tree of tasks. This is a binary search tree [64, 65]; the waiting time is used to order the tasks.

4.6 Scheduler Tuning Experiment Design

The hardware used was a laptop with Intel Core 2 Duo T8300 CPU (2.4GHz and 3MB L2 Cache) and 2GB RAM. The following three main “knobs” were selected to tune the scheduler: A, minimum granularity, B, latency, and C, wakeup granularity.

The parameters are specified in nanoseconds. A and B range from 100,000 to 1,000,000,000. C ranges from 0 to 1,000,000,000. The ranges were discovered by examining the `sysctl.c` source file in the Linux source tree, and then verified from a terminal command line. The default values of these parameters are: A=8,000,000, B=40,000,000, and C=10,000,000 (all in nanoseconds). Maurer [56] discusses what these parameters are used for: The latency value is a time period during which all tasks must use the CPU at least once. Latency is divided by the minimum granularity to give the number of tasks supported. If the number of tasks exceeds this, the latency period is extended automatically. The wakeup granularity is a grace period that a task could have on the CPU before being preempted. Our response is the turnaround time for the workload, which is directly related to the throughput: the more runs of the workload complete per unit time, the faster the workload completes.

A Central Composite Design (CCD) was selected, because of its power in detecting and modelling curvature. Twenty runs were required according to the CCD model [62]. There were 3 factors (A, B and C, as defined above). A quadratic model was selected. This setting gave favourable indicators of a good model. These indicators were:

1. The degrees of freedom for the lack of fit and pure error were 5, indicating a reliable lack of fit test.
2. Standard errors were similar within the type of coefficient i.e. for (A,B, and C), (AB, AC, and BC), and (A^2 , B^2 , and C^2), which is required for a good model [62].
3. R-squared is 0.00 for A,B,C, AB, AC, and BC, and small for A^2 , B^2 , and C^2 , which is required for a good model [62].
4. Leverages for most runs were close to 0.00, which means small errors in measurement will have limited impact on the model.

4.7 Scheduler Tuning Model Analysis

The response in the model was transformed using a power transformation $y' = (y+k)^\lambda$, where $\lambda = -1.99$, $k = 0$. This was necessary because the model produced had significant lack of fit. In such a case, a Box-Cox plot would show that the minimum model residual (Residual SS) falls outside of the 95% interval. The power transformation works by evaluating the model in order to find a value of λ which would cause the minimum model residual to fall within the 95% interval. An insignificant lack of fit is then achieved. Figure 4.3 shows the Box-Cox plot of the transformed model, with the minimum residual falling within the desired 95% interval, indicated by the λ just to the right of -2 .

Figure 4.4 shows the response cube, with the various response values for different combinations of the three factors in the model. Figure 4.5 shows the normal plot of residuals for the model. It is desirable for the points to be in line, such that one can cover all of them with a pencil, if they were printed. This indicates that the residuals are normally distributed. Figure 4.6 shows a plot of residuals versus predicted values. This shows the desired uniform spread, with no outliers, which would have been indicated by points beyond either one of the two horizontal red lines. Figure 4.7 shows the residuals

versus run number plot. This figure shows the desired spread, with no obvious pattern, implying that there was no bias in the design of the experiment, in terms of the order of runs. Figures 4.8, 4.9, and 4.10, show plots of residuals versus each one of the three factors. There is a uniform spread of the residuals, which is what is desired. It means that there is constant variance of residuals for different levels of the factors.

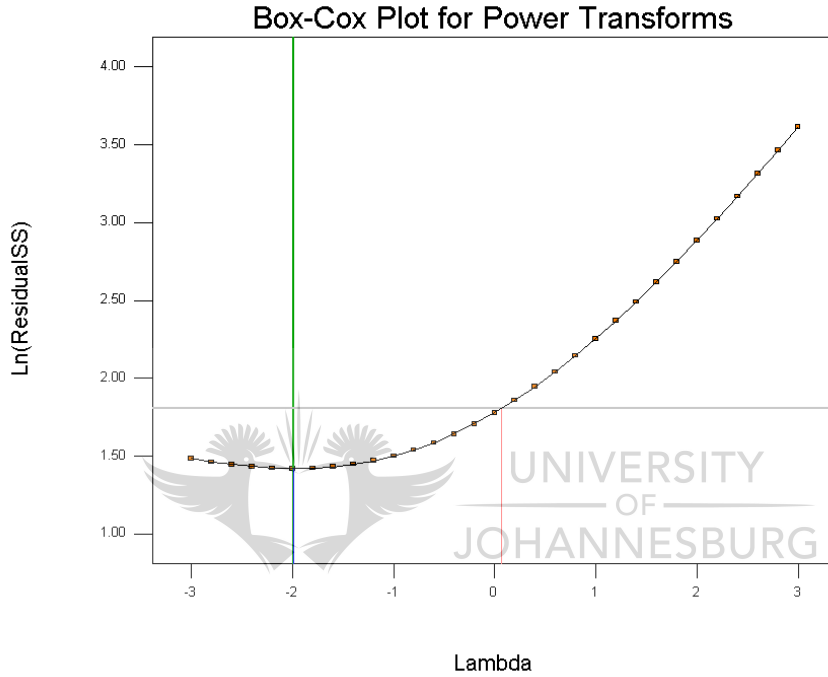


Figure 4.3: Box-Cox plot for model

Tables 4.1 and 4.2 show details of our model's lack of fit and ANOVA tests. The quadratic model had the most significance (p-value < 0.0001) when compared to other orders, none of which were significant. The model did not have significant lack of fit (p-value of 0.0965). This means that there is a relatively large chance (9.65%) that the lack of fit value could be due to noise [45, 61, 62]. Parameter C and C^2 were found to be the significant model terms (p-value < 0.0001). The predicted R-squared (0.9264) and adjusted R-squared (0.9855) values are close to 1 and support each other, which is desirable [62]. The adequate precision of the model is greater than 4 (it is actually 29.366), which suggests an adequate signal, meaning that the model can be used to navigate the design space [62]. Adequate precision is a measure of the experimental signal-to-noise ratio. Adequate precision is derived by comparing the range of the

4.7 Scheduler Tuning Model Analysis

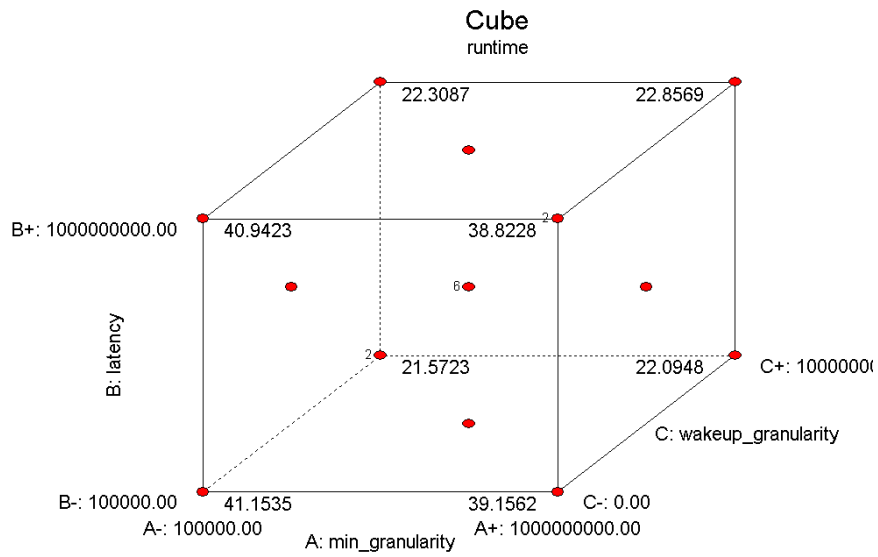


Figure 4.4: Response cube

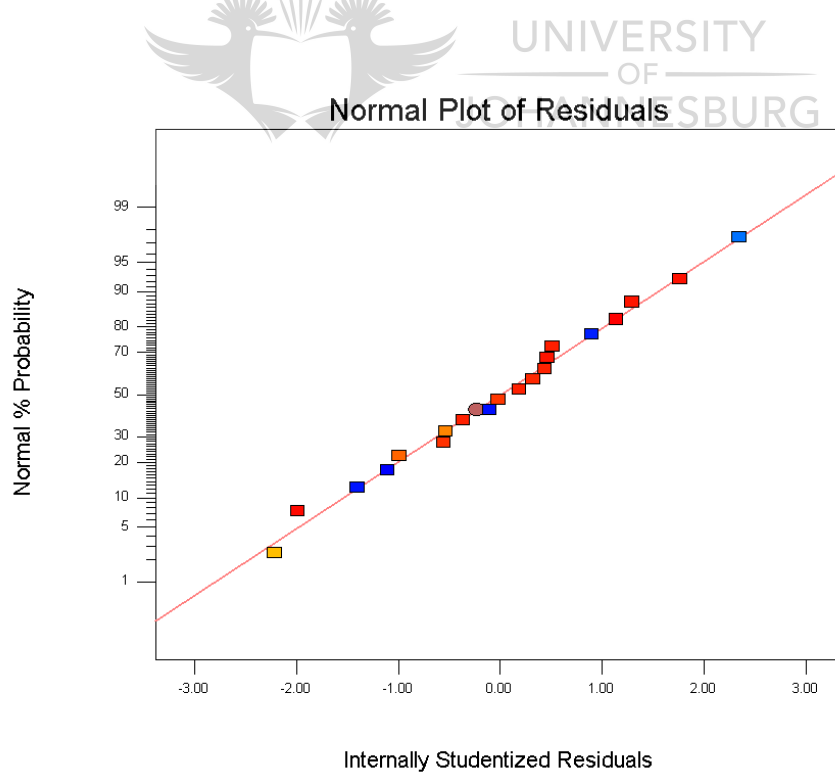


Figure 4.5: Normal plot of residuals

4.7 Scheduler Tuning Model Analysis

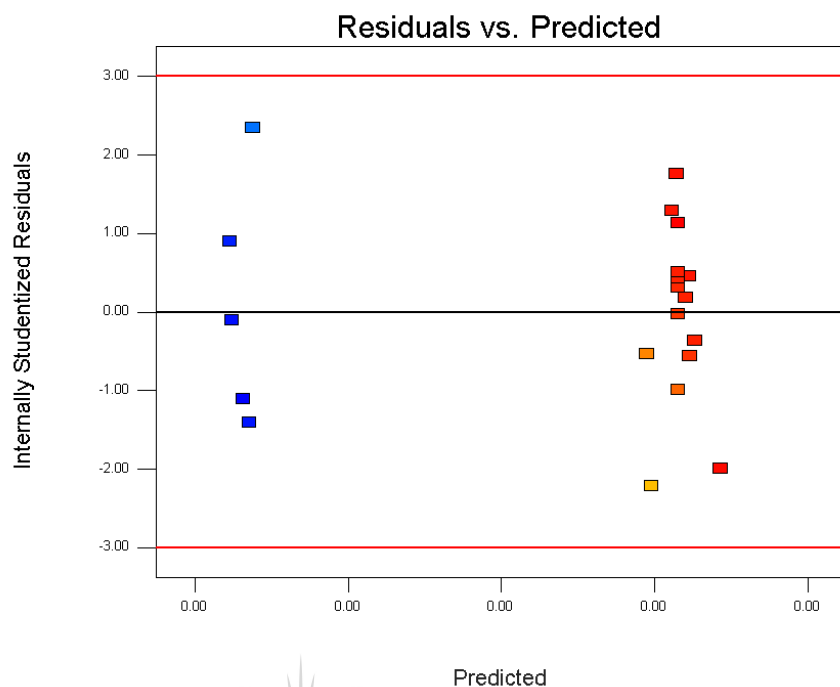


Figure 4.6: Residuals versus predicted plot

predicted values at the design points to the average variance of the prediction. The final equation for the model is shown in Equation (4.3).

Table 4.1: Model lack of fit summary

Source	Sequential p-value	Lack of Fit p-value	Adjusted R-Squared	Predicted R-Squared
Linear	0.0004	< 0.0001	0.6095	0.4477
2FI	0.9876	< 0.0001	0.5241	-0.9090
Quadratic	< 0.0001	0.0965	0.9855	0.9264
Cubic	0.0993	0.1953	0.9923	0.0704

4.7 Scheduler Tuning Model Analysis

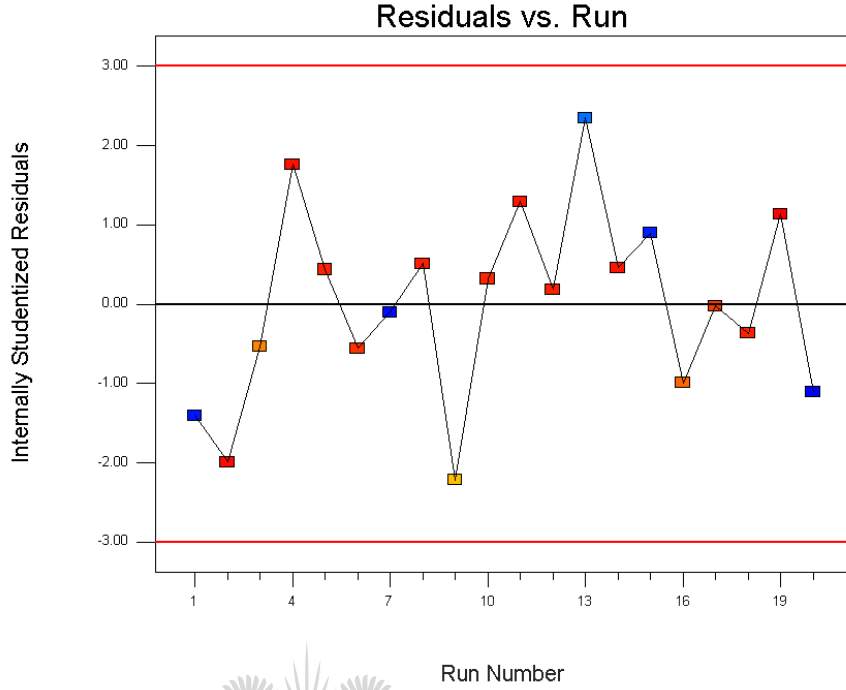


Figure 4.7: Residuals versus run number plot

$$\begin{aligned}
 runtime^{-1.99} &= 0.0020771967 \\
 &- 8.4916937232645E-6 \times A \\
 &- 3.2892310822324E-5 \times B \\
 &+ 0.0007224509 \times C \\
 &+ 1.3253026076439E-6 \times AB \\
 &- 4.1705431190103E-5 \times AC \\
 &- 3.7371875889965E-5 \times BC \\
 &+ 4.6223091363884E-5 \times A^2 \\
 &- 5.3567800702584E-5 \times B^2 \\
 &- 0.0006982174 \times C^2
 \end{aligned} \tag{4.3}$$

The runtime is the turnaround time, which is proportional to the throughput. The

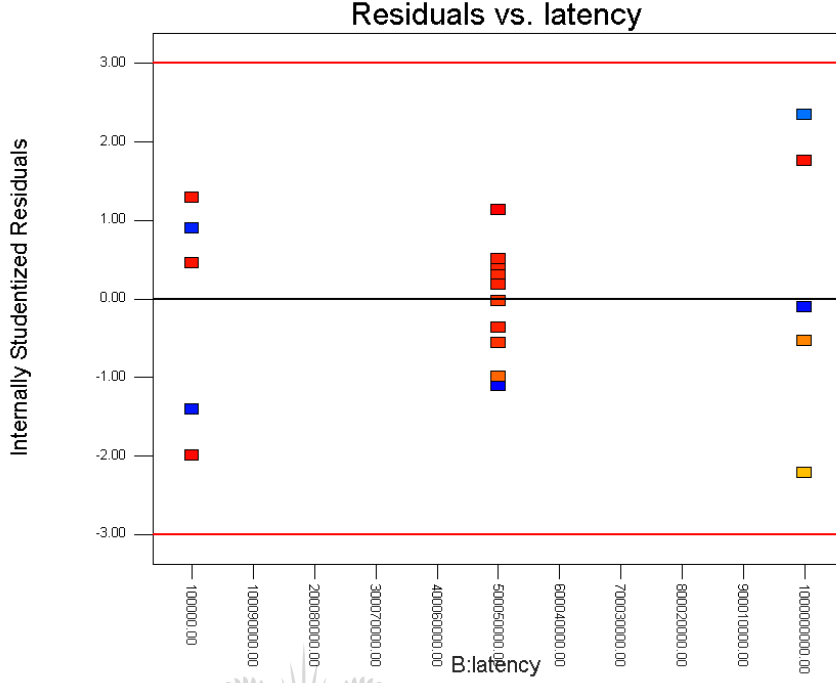


Figure 4.8: Residuals versus Latency plot

throughput is the response we are trying to optimize.

$$\text{Throughput} = \frac{\text{Number of Tasks Completed}}{\text{Time Taken}} \quad (4.4)$$

4.8 Experiment and Results

This section discusses the experiment environment and tools and the results. Our methodology is in line with that described by Jain [25], as mentioned in section 2.2. We instrument the system by observing the turnaround time; we monitor by measurement; there is no need to characterize the workload because it is known; we predict performance by tuning the scheduler using various settings, as specified by the RSM model, and measuring the performance, and then we optimize to obtain the best performance alternative (see Equation 4.4).

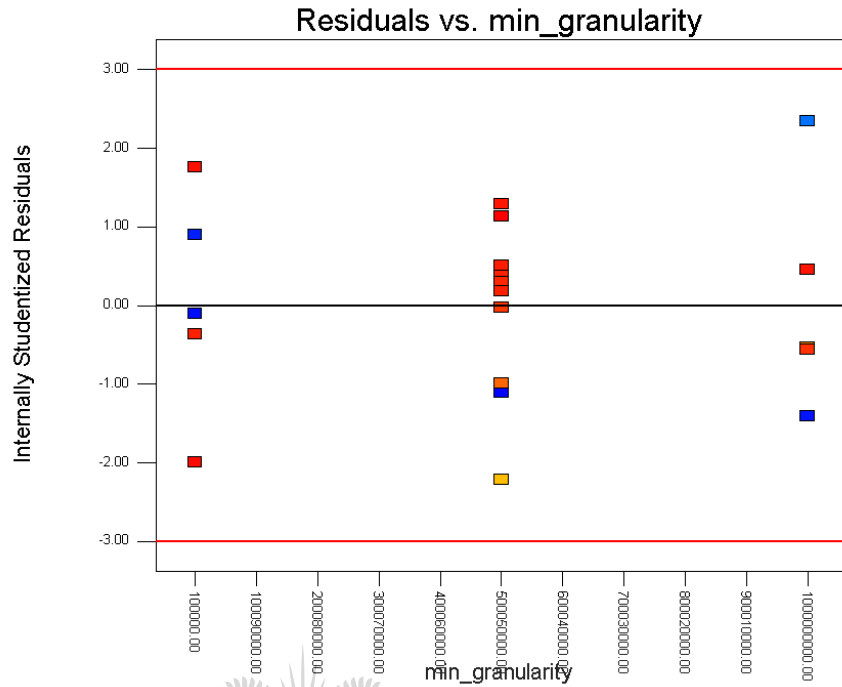


Figure 4.9: Residuals versus Minimum Granularity plot

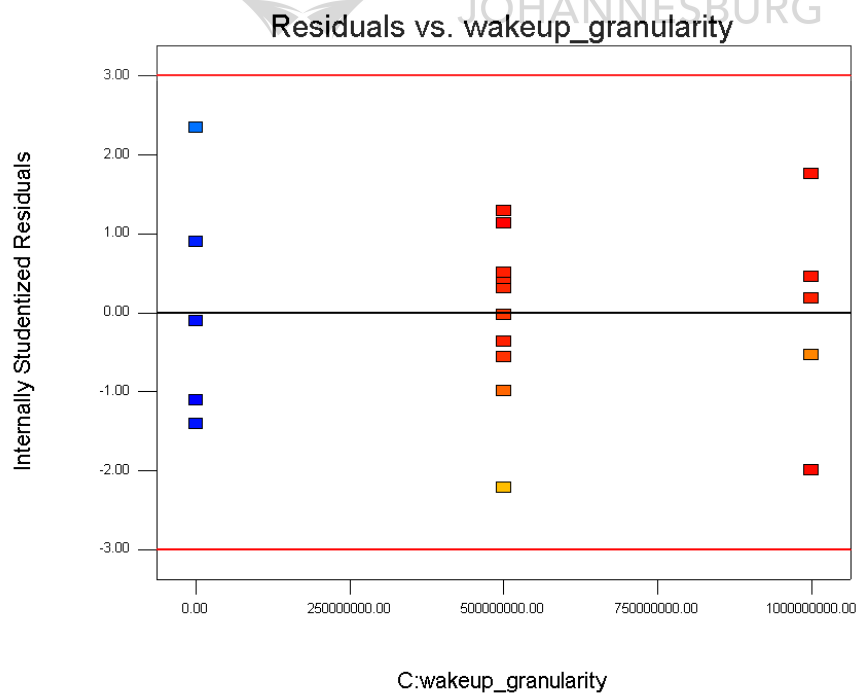


Figure 4.10: Residuals versus Wakeup Granularity plot

Table 4.2: ANOVA for response surface quadratic model

Source	Sum of Squares	Degrees of Freedom	Mean Square	F Value	p-value Prob > F
Model	7.734E-6	9	8.594E-7	144.196	< 0.0001
A (min granularity)	7.211E-10	1	7.211E-10	0.121	0.7352
B (latency)	1.082E-8	1	1.082E-8	1.815	0.2076
C (wakeup granularity)	5.219E-6	1	5.219E-6	875.753	< 0.0001
AB	1.405E-11	1	1.405E-11	0.002	0.9622
AC	1.391E-8	1	1.391E-8	2.335	0.1575
BC	1.117E-8	1	1.117E-8	1.875	0.2009
A^2	5.876E-9	1	5.876E-9	0.986	0.3442
B^2	7.891E-9	1	7.891E-9	1.324	0.2766
C^2	1.341E-6	1	1.341E-6	224.946	< 0.0001
Residual	5.960E-8	10	5.960E-9		
Lack of Fit	4.643E-8	5	9.287E-9	3.526	0.0965
Pure Error	1.317E-8	5	2.634E-9		
Cor. Total	7.794E-6	19			

4.8.1 Benchmark Experiment

In order to measure the performance of an operating system component such as a scheduler, a workload has to be used. This workload is in the form of a benchmark. As discussed by Feitelson [66], a micro-benchmark measures performance of specific system components. The benchmark we selected is hackbench [57], which has been used by experts to measure the performance of the CFS Linux scheduler [47, 48, 67]. Hackbench is part of a set of micro-benchmarks used by the creator of the CFS Linux scheduler to compare its performance to other schedulers [57]. Hackbench measures the performance, overhead, and scalability of the scheduler [68]. It works like a chat application, in which there are several groups of processes, with each group having servers and clients. Each client sends a message to every server in the group. Hackbench was also used in other scheduling studies [69, 70, 71, 72, 73, 74].

Hackbench has three parameters that can be set to control its behaviour [46]: to control whether the scheduling entities are threads or processes; to control how many groups of scheduling entities are created to communicate with one another; and how

many times each entity loops (communicates with the other entities in its group). Each group has 40 threads/processes. A process is an instance of a program that is being executed while a thread is a flow of execution within a process [75]. The default number of loops is 100. We initially used a configuration of 5 groups of threads, with 10,000 messages sent between each pair of processes in a group, then increased the number of groups. We chose threads instead of processes because threads are used in modern server workloads such as web servers and database servers [76]. Threads share parts of memory and some resources, with less overhead required for creating and running them, and are therefore called *lightweight processes* [3, 2, 4, 75]. Threads offer better performance than processes, when used to implement applications. An application implemented as a set of 100 threads in a single process will have shorter runtime than one implemented as a set of 100 separate processes [75]. We chose 5 groups with 10,000 messages because that generated a sufficiently large workload to detect difference in performance. We experimented with smaller numbers of messages, which is a smaller workload, and this produced run times for workloads running on the optimized and unoptimized systems which did not have large differences. When comparing two such systems in an experiment, one has to use a sufficiently large workload to observe differences, and we found this to be 10,000 messages and 5 groups. Larger workloads will show greater differences. Figure 4.12 shows what happens as the number of groups is varied; as the workload increased (larger number of groups) our method shows a larger difference in performance results between the optimized and unoptimized systems. Using such large workloads is important because of the fast CPU used, in order that differences can be recorded. Memory was not an issue, since there was more than enough (4GB RAM); hackbench does not use large messages for communication, and has low memory requirements. Memory would become a problem only if a very large number of threads was used. We used enough threads to demonstrate the usefulness of our approach. This was controlled by varying the number of groups, since there are 40 threads per group in hackbench.

The benchmark environment was an Ubuntu 9.04 Linux laptop, running kernel version 2.6.28, running in a console (command line interface) with the GUI disabled. Screen blanking was also turned off. This ensures that there is minimal work for the scheduler coming from processes other than the hackbench workload. To test for this stability, hackbench was run in this environment with 5 groups of 40 threads, 10000

loops, and 100 runs, twice. The average turnaround time for both sets of runs was approximately the same with a very similar and low standard deviation of 0.19. This shows the test environment is stable. Design, analysis and optimization were done using Design Expert 7 [77].

When measuring metrics like turnaround time, normally several runs are required for each configuration [25]. The formula for finding the number of runs, i.e. the sample size needed to determine a mean, as given by Jain [25] is shown in Equation (4.5) below:

$$n = \left(\frac{100zs}{r\bar{x}} \right) \quad (4.5)$$

where z is the normal variate of the desired confidence level (we chose 95% confidence), s is the standard deviation, r is the accuracy (we chose an accuracy of approximately 2%, which provides a good balance between accurate results and short benchmarking time), and \bar{x} is the mean of the turnaround times. The choice of 95% confidence was made because this level indicates very high confidence in experiment results and is standard [25, 78]. We first executed the benchmark for 10 runs, which allowed for the determination of the above values. The optimized runs showed a small standard deviation, while the unoptimized showed a larger standard deviation. Unoptimized benchmarking requires 12 runs, while optimized required just one, but we used 12 for optimized also for consistency.

Another experiment was carried out to compare the scalability of the optimized and the unoptimized scheduler. In this experiment, the load varied from 1 hackbench group of 40 threads to 20 groups (800 threads). For each group size, 12 runs were executed, and the mean calculated. The mean was then used as the runtime value for a particular group size in the RSM process.

4.8.2 Results of Experiment (Optimization)

To optimize the response, we need to minimize the turnaround time. The following are the optimal values for the parameters: $A \approx 103600$, $B \approx 242437236$, $C \approx 780657445$. Optimization was done in Design Expert, using the Simplex method [62]. For these values, the response is predicted to be 20.922 for the first experiment. Figure 4.11 shows a 3D plot of the transformed response against the three factors.

4.9 RSM-Based Parameter Tuning Methodology

The benchmark was run with the default Linux CFS scheduler parameters (12 runs). The average runtime was 24.91 seconds, with a standard deviation of 2.89. The benchmark was then run again with the optimal parameter settings described above, also for 12 runs. The average runtime was 22.09 seconds, with a standard deviation of 0.19. This represents an 11% improvement. It should be noted that the hackbench settings used were for a small workload, therefore the runtime values generated are also small. This was an initial test.

Figure 4.12 shows the results of the scalability test. The x axis indicates the number of hackbench groups used, i.e. the load, while the y axis shows the turnaround time. With the scalability test, the maximum gain is obtained from the hackbench configuration of 20 groups. There is a 16.48% improvement in the optimized scheduler as opposed to the default scheduler. This improvement will be possible with all workloads which do not depend on a fast response time, for example for user interaction, but rather throughput. The optimized scheduler clearly performs better (shorter turnaround time) and also scales better; note the almost-straight line as the load increases. The line for performance of the unoptimized scheduler with the same workload is not as straight, showing quite a few deviations, and turnaround time is higher, as also indicated in the first experiment. The cause of the uneven line for the unoptimized scheduler is that its performance degrades earlier, as the number of processes increases. With the default settings, the scheduling latency parameter (B) has a relatively small value (compared to the optimized scheduler) and minimum scheduling granularity parameter (A) is relatively large (compared to the optimized scheduler), meaning that threads are not given much time on the CPU. The scheduling wakeup granularity parameter (C) tries to ensure that a task is not preempted too soon, so can result in extending a task's time on the CPU. In the optimized kernel this value is larger, so tasks have more time with the CPU to finish what they are doing. The combination of these settings means that larger workloads used cannot be scheduled predictably by the unoptimized CFS scheduler, since tasks are not given enough time on it, leading to the uneven line in figure 4.12.

4.9 RSM-Based Parameter Tuning Methodology

Algorithm 4.1 shows a possible methodology for tuning the operating system scheduler.

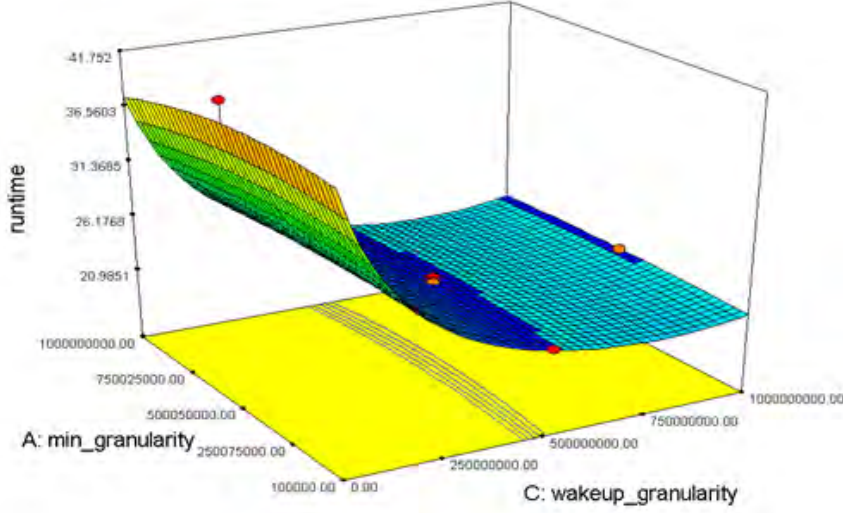


Figure 4.11: 3D response surface for runtime (latency is constant with a value of 500050000)

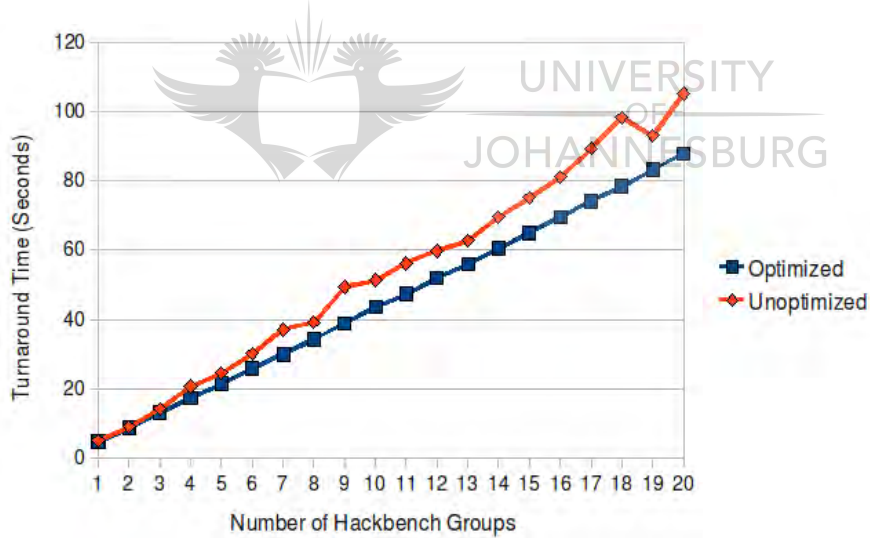


Figure 4.12: Hackbench turnaround time for the optimized and unoptimized Scheduler

4.10 Conclusion

Our results show that the RSM approach to scheduler tuning results in better performance compared to using unoptimized scheduling parameters. The values for the parameters are obtained after a limited number of experiments. RSM provides a mathematical basis for choosing parameter settings, which is scientific, as opposed to an

Algorithm 4.1 RSM-based operating system scheduler tuning methodology

- 1: Get workload
 - 2: Get scheduler tuning parameter ranges
 - 3: Design experiment using RSM
 - 4: Execute experiment
 - 5: Obtain model linking runtime metric with parameter values
 - 6: Optimize model
 - 7: Get optimal parameter values
 - 8: Run workload in production environment with optimal parameter values
-

approach based on rules of thumb used by system administrators, where experience is very important. With RSM, even systems staff with little experience can optimize the performance of the scheduler. One can see that optimization using RSM leads to very stable execution of the workload used (small standard deviation) with excellent scaling (performance degrades gracefully as the workload increases). Therefore, not only is the turnaround time shorter for a given workload, but also performance is predictable, making this solution ideal for applications which involve increasing loads. It should be noted that at present our tool requires a workload that can finish execution in good enough time for optimal parameter settings to be discovered. Greater benefits will be obtained if the workload increases within the same application. For example, if comparing two systems, one using the unoptimized scheduler and the other one using the optimized scheduler, and they both have workloads like the one used that are increasing at the same time on both systems. The larger the load, the greater the difference will be between both systems. The next chapter continues with exploration of optimization methods for operating system tuning, this time using standard optimization methods.

Chapter 5

Application of Standard Optimization Methods to Operating System Scheduler Tuning



5.1 Summary

This chapter continues where the previous chapter left off, which was mainly looking at experimental design methods to obtain scheduler tuning solutions. It focuses on standard optimization methods, in an attempt to build the knowledge base in this area. This chapter describes a study of comparison of application of global and one-dimensional local optimization methods to operating system scheduler tuning. We are optimizing the throughput (Equation 4.4). The Operating System scheduler we use is the Linux 2.6.23 Completely Fair Scheduler (CFS) running in a simulator (LinSched). We have ported the Hackbench scheduler benchmark to this simulator and use this as the workload. The global optimization approach we use is Particle Swarm Optimization (PSO) [79]. We make use of Response Surface Methodology (RSM) [62, 78] to specify optimal parameters for our PSO implementation. The one-dimensional local optimization approach we use is the Golden Section method [81, 82]. In order to use this approach, we convert the scheduler tuning problem from one involving setting of three parameters to one involving the manipulation of one parameter. Our results show

that the global optimization approach yields better response but the one-dimensional optimization approach converges to a solution faster than the global optimization approach.

This chapter is organized as follows: Section 5.2 gives an overview of the problem; Section 5.3 gives some background on research in optimization of software; Section 5.4 gives an overview of scheduling in the operating system, using Linux as an example; Section 5.5 discusses operating system scheduler simulation; Section 5.6 discusses our experiments involving application of optimization methods to improving the performance of the Linux operating system process scheduler, involving both global optimization and local optimization; Section 5.7 discusses our results; Section 5.8 presents the standard optimization-based tuning methodology, and Section 5.9 has our conclusion.

5.2 Scheduler Tuning Introduction

Operating systems play the role of resource managers: modern computer systems consist of many resources such as main memory, hard disk drives and other storage devices, network interfaces, and the CPU [2, 3, 4]. Operating Systems consist of several components, each being responsible for managing a subset of the available resources. In this chapter, we are concerned with optimizing the operating system scheduler by tuning its parameters. The job of the operating system scheduler is to decide which task should run on the CPU next, and for how long. Algorithm 5.1 shows how a simple scheduler works. The criterion for selecting the next task could be priority (some tasks are more important than others) or round-robin, for example. Programs running on computer systems are represented as tasks, which keep track of allocated memory and various types of state. For a task to do its work, it must use the CPU i.e. the instructions that make up the task must be *executed*, or obeyed. A collection of these tasks running on a computer makes up a *workload*. Various metrics are used to assess the performance of a computer system. One is response time: operating system designers would like to make sure the time between when the user makes a request and it is handled is minimized. Another metric, and the one we are interested in, is turnaround (or completion) time. We want to minimize the time it takes a workload to finish its work.

In this chapter we study the Linux scheduler (known as the Completely Fair Scheduler) and apply global optimization (Particle Swarm Optimization, after meta-optimization)

Algorithm 5.1 Simple Scheduling Algorithm

Require: $P \Leftarrow$ List of ready tasks, $\{p_0 \dots p_n\}$

```

1: loop
2:   Select task  $\tau$  from  $P$  based on criterion, such as priority
3:   Assign task  $\tau$  to CPU to run for time  $T$ , remove  $\tau$  from  $P$ 
4:   if  $\tau$  blocks then
5:     Add  $\tau$  to List of blocked tasks  $B$ 
6:   else if  $\tau$  reaches time period  $T$  then
7:     Return  $\tau$  to  $P$ 
8:   else if  $\tau$  terminates then
9:     Free resources allocated to  $\tau$ 
10:  end if
11: end loop

```

and local optimization (Golden Section method) to tuning it. Weise [79] categorizes Particle Swarm Optimization as an example of global optimization, and this is the classification we use. Note that others, for example, Solis and Wets [80] use a more stringent definition for global optimization, and do not put basic PSO in this category. Golden Section, as described by Venkataraman [81] and Chong and Zak [82] is an approach for local optimization, stopping once a minimum is reached.

We decided to use global optimization because our search space is quite large; the ranges of the three scheduling parameters are quite large, so there is a danger of encountering local optima. PSO was selected because it has been shown to outperform other popular global optimization methods, such as Genetic Algorithms [83, 84, 85] and Ant Colony Optimization [84], and Simulated Annealing [85], for similarly structured problems, in terms of computational efficiency, robustness, and optimality. Golden Section used because of its efficient solution of one-dimensional problems.

5.3 Optimization in Software Systems

The previous chapter discusses experiments which use a statistical optimization approach known as Response Surface Methodology (RSM) to tune the Linux operating system kernel. The RSM approach involved designing a limited number of experiments; each experiment involved setting each of three scheduling parameters. A benchmark is

then run for each experiment. Based on the performance results, a model was obtained. This model was then optimized in order to give the best setting for each one of the three parameters. We achieved a 16.48% performance improvement when using the optimal parameter settings over when the Linux scheduler's default parameter settings were used. The approaches described in this chapter shed new light on use of standard global and local optimization for tuning the operating system scheduler.

5.4 Operating System Scheduling in Linux

The job of the operating system scheduler is to decide which task should run on a CPU next and for how long [2, 3, 4]. Schedulers implement scheduling algorithms, which could have goals of minimizing metrics such as waiting time (the time tasks spend waiting to use a CPU), response time (the time between when an event occurs and when it is handled, in others attended to by the operating system), and maximizing metrics such as throughput (tasks completed per given period of time) and fairness (ensuring that each task gets a fair share of the CPU).

The Linux kernel, starting from version 2.6.23, makes use of a scheduler known as the Completely Fair Scheduler (CFS). Kumar [55] and Maurer [56] describe the internals of the CFS. The Linux CFS scheduler keeps track of how much tasks are being treated unfairly. A task is being treated unfairly if it is not being executed by the CPU. It maintains a binary search tree (a red-black tree) that orders tasks according to how unfairly they are being treated. A task that has been treated the most unfairly (because it has been waiting to use the CPU for a long time) is selected to execute on the CPU at the next opportunity. The CFS scheduler is modular, making it readily extensible, and also supports group scheduling and scheduling classes.

5.5 Operating System Scheduler Simulation

LinSched is a Linux scheduling simulator developed by Calandrino et al. [51]. The code related to the Linux scheduler was extracted from the Linux 2.6.23 kernel and modified, so that it runs as an ordinary process. The process also consists of a simulation-management component, which runs in a loop, until the maximum number of ticks has expired. When a task (process or thread) is running on a regular Linux kernel, each

5.5 Operating System Scheduler Simulation

time it is scheduled, i.e. assigned to a CPU, its instructions are executed, continuing where they left of last time it was scheduled. After the process has acquired the CPU for a long-enough combined period, it exits. With LinSched, a function is declared to represent a particular type of task. Every time the task is assigned to use a CPU, the function is called.

LinSched has many advantages for use as a research tool, over a kernel. One is that the scheduler can be modified and tested easily, since it runs as a user-level process. A bug in the scheduling code causes only the LinSched process to crash. Working with an actual kernel requires more time because the kernel has to be compiled then rebooted. A bug could bring down the kernel together with all the other processes that are running. Then the cycle of rebooting to a trusted kernel, fixing the bug, and rebooting again continues. LinSched also allows fast validation of scheduling algorithms, since the scheduling sub-system is isolated. In our research we have chosen to use the LinSched simulator because it is the only one available that is based directly on actual Linux code, while completely isolating the scheduling subsystem. This was made clear after an extensive literature review. Other simulators, e.g. AKULA [12], require the Linux scheduler's code to be ported to them.

We have ported the hackbench [46] scheduling benchmark to LinSched. This means that we have added modified hackbench code to the LinSched code. Therefore, the tasks LinSched creates are actually hackbench threads. The hackbench benchmark simulates a chat application with a group of 20 senders and 20 receivers; these values are hard-coded in the benchmark. Each sender sends a message to each of the receivers, who receive from all the senders. The number of messages to be sent can be specified, as can the number of groups in the system, thus enabling the user to vary the workload. With our addition to LinSched, each sender is represented by a special "linsched_writer_callback" function and each receiver is represented by a special "linsched_reader_callback" function. These functions are implemented as finite state machines, so that if a sender is scheduled, for example, i.e. its callback is invoked, it continues where it left off last time it used the CPU. This means that it remembers its state. We have validated the simulator by running increasing sizes of hackbench workloads (5 groups of hackbench threads with varying number of messages to be sent by each sender). The choice of 5 groups was made because with just one group, the performance difference is not very pronounced. This logically follows from the fact that

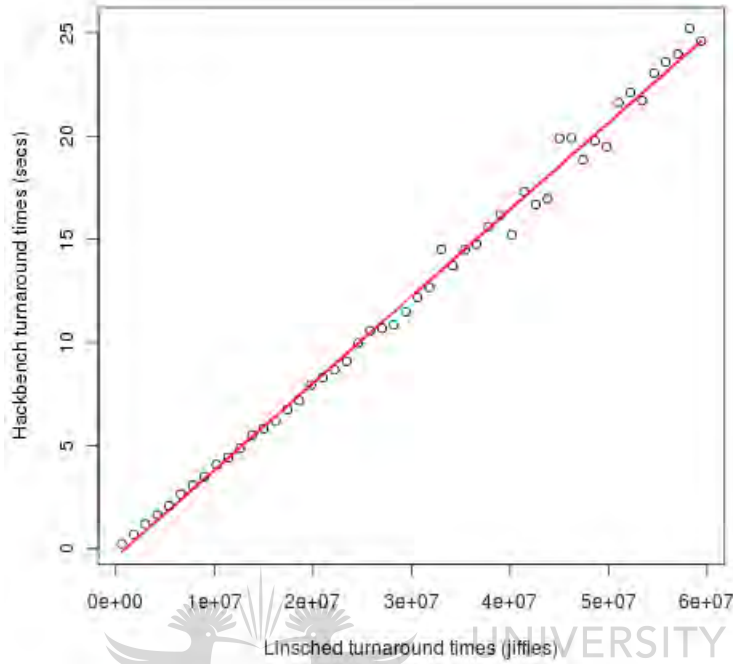


Figure 5.1: Linear Relationship Between LinSched Simulator Jiffies Metric and Hackbench Seconds Metric

if a benchmark's workload is minimal, it does not make much of a difference what the scheduler parameter settings are. For each workload, we record the runtime when it runs on the LinSched simulator, and when it runs on a regular Linux installation. The relationship is linear, as is shown in Figure 5.1. The runtime is giving in Jiffies, which are roughly equivalent to milliseconds (1000 Jiffies in one second).

5.6 Optimization of the Operating System Scheduler

Here we describe two approaches to tuning the Linux CFS scheduler. This scheduler has several parameters that can be set to tune its operation in order to achieve the best performance for a particular workload. We focus on three parameters, specified in ns. They are:

- Latency: A period of time during which all tasks should use a CPU. Range is 100,000 to 1,000,000,000 ns.

5.6 Optimization of the Operating System Scheduler

- Minimum Granularity: The minimum period of time a task should have on a CPU. Range is 100,000 to 1,000,000,000 ns.
- Wakeup Granularity: An additional allowance; a task is allowed to use a CPU for this amount of time, even after it has exhausted its primary allowance on the CPU. Range is 0 to 1,000,000,000 ns (nanoseconds).

5.6.1 Global Optimization

We make use of the basic Particle Swarm Optimization algorithm, as described in [49, 82]. In this version, there are a set of particles, each of which has a certain location, x and a velocity v . The formula for updating a particle's velocity is:

$$\vec{v} \leftarrow \omega \vec{v} + \phi_p r_p (\vec{p} - \vec{x}) + \phi_g r_g (\vec{g} - \vec{x}) \quad (5.1)$$

The formula for updating a particle's position is:

$$\vec{x} \leftarrow \vec{x} + \vec{v} \quad (5.2)$$

Equations (5.1) and (5.2) are as given in [49]. Note that in our implementation of PSO, r_p and r_g differ for each dimension of the vectors involved. The particles move through the search space updating, keeping track of their best position, as well as the swarm's best position. In equation (5.1) ω is the inertia weight; ϕ_p is the cognitive parameter and ϕ_g is the social parameter; ϕ_p is set according to the weight the cognitive component should have; ϕ_g is set according to the weight the social component should have; r_p and r_g are random numbers uniformly distributed in $[0, 1]$; p is the best position of a particle and g is the best global position.

We used meta-optimization [49] to come up with the ω , ϕ_p and ϕ_g parameters. Our meta-optimization approach makes use of RSM [45, 62]. The tool we made use of is the *rsm* [86] package in the R statistical application [87]. We first of all designed an experiment for our three-parameter problem, using a Box-Behnken design. This resulted in a randomized experiment of 14 runs, with differing combinations of parameters. The ω parameter took on the values 0, 0.45, or 0.9. The ϕ_p and ϕ_g parameters took on the values 0, 2, or 4. Our PSO configuration had 20 particles and a maximum of 30 iterations. We came up with these numbers (20 and 30) through exploratory experiments using PSO together with hackbench (empirical search). These numbers allowed for

5.6 Optimization of the Operating System Scheduler

Table 5.1: Meta-optimization: RSM-designed PSO experiment runs

Run	ω	ϕ_p	ϕ_g	R^*	I^*	$rsum = R^* + I^*$
1	0	0	2	4294667318	5	4294667323
2	0.9	0	2	4294667318	4	4294667322
3	0	4	2	4294667318	29	4294667347
4	0.9	4	2	4294667318	30	4294667348
5	0	2	0	4294668048	31	4294668079
6	0.9	2	0	4294669150	31	4294669181
7	0	2	4	4294667315	7	4294667322
8	0.9	2	4	4294667315	6	4294667321
9	0.45	0	0	4294669038	31	4294669069
10	0.45	4	0	4294668498	31	4294668529
11	0.45	0	4	4294667315	5	4294667320
12	0.45	4	4	4294667315	5	4294667320
13	0.45	2	2	4294667315	18	4294667333
14	0.45	2	2	4294667315	16	4294667331

sufficient variation in particle behavior before convergence. Table 5.1 shows the design of the experiments as well as the results. The value of R^* is the minimum response attained for that particular run i.e. the smallest turnaround time. The value of I^* is the number of iterations required to converge to the minimum response (the reader is reminded that our PSO algorithm uses 20 particles and a maximum of 30 iterations). The metric $rsum = R^* + I^*$ is used to compare the various runs. The right-most three columns will be discussed in Section 5.7, the results section.

5.6.2 Local Optimization

We make use of Golden Section one dimensional optimization, as described in [82]. As discussed earlier, the Linux CFS scheduler has three parameters for tuning we are concerned with: latency, minimum granularity, and wakeup granularity. We name these *Latency*, *MinGran*, and *WakeupGran*, respectively. Within the CFS scheduling algorithm, another important variable is the *scheduling period* [55, 56]. This is actually the period of time during which all tasks should run once. We call this *SchedPeriod*. We name number latency, the maximum number of tasks that can be handled in *Latency*,

5.6 Optimization of the Operating System Scheduler

NrLatency. The following equations are involved in calculation of the scheduling period.

$$NrLatency = \frac{Latency}{MinGran} \quad (5.3)$$

If the number of tasks running, *NumTasks*, is such that,

$$NumTasks \leq NrLatency \quad (5.4)$$

then

$$SchedPeriod = Latency \quad (5.5)$$

Otherwise, we have the following:

$$SchedPeriod = Latency \times \frac{NumTasks}{NrLatency} \quad (5.6)$$

In order to apply the Golden Section, we convert the optimization problem from one depending on three variables, *Latency*, *MinGran*, and *WakeupGran*, into one depending on one variable, *SchedPeriod*. This is done by making use of the relationships between the various variables evident in the Linux kernel source code. So, instead of manipulating the values for the three variables, we manipulate only the *SchedPeriod* variable, and for each value of this variable, we discover the required values for *Latency*, *MinGran*, and *WakeupGran*, required to give this value. There are two possible discovery algorithms. Algorithm 5.2 shows the first Parameter Discovery Algorithm. This works for the case when *NumTasks* is greater than *NrLatency*. In other words, there are many tasks running. Here *Latency* is set at 19,900,000, the maximum required in order to use this approach. The value of 200,000,000,000 for *SchedPeriod* governs what values will be assigned to *WakeupGran* and *MinGran*. This relationship was discovered by examining the Linux kernel source code and, with our data, calculating thresholds which govern the assignment of values to the variables. We know the number of tasks in the hackbench workload: 5 groups of 40 tasks = 200 tasks, since there are 20 senders and 20 receivers in each group.

Algorithm 5.3 shows the second Parameter Discovery Algorithm. This works for the case when *NumTasks* is less than or equal to *NrLatency*. In other words, there are relatively few tasks running. Here *Latency* is the same as *SchedPeriod*. If the *Latency* is small ($< 20,000,000$), *MinGran* depends on *SchedPeriod* and *NumTasks*, otherwise it is set at 100,000.

5.6 Optimization of the Operating System Scheduler

Algorithm 5.2 Parameter discovery algorithm 1, used by Golden Section 1

```
1: if SchedPeriod > 200,000,000,000 then
2:   WakeupGran  $\leftarrow$  SchedPeriod - 200,000,000,000
3:   SchedPeriod = 200,000,000,000
4: else
5:   WakeupGran  $\leftarrow$  0
6: end if
7: Latency  $\leftarrow$  19,900,000
8: MinGran  $\leftarrow$  SchedPeriod/200
9: return Latency, MinGran, WakeupGran
```

Algorithm 5.3 Parameter discovery algorithm 2, used by Golden Section 2

```
1: if SchedPeriod > 1,000,000,000 then
2:   WakeupGran  $\leftarrow$  SchedPeriod - 1,000,000,000
3:   SchedPeriod  $\leftarrow$  1,000,000,000
4: else
5:   WakeupGran  $\leftarrow$  0
6: end if
7: Latency  $\leftarrow$  SchedPeriod
8: if Latency < 20,000,000 then
9:   MinGran  $\leftarrow$  SchedPeriod/200
10: else
11:   MinGran  $\leftarrow$  100,000
12: end if
13: return Latency, MinGran, WakeupGran
```

The Golden Section search optimization method [81, 82] involves considering the x axis and two extreme points on this axis, a_0 and b_0 . Two inner points (a_1 and b_1 , to the right of a_1) are evaluated empirically, using the hackbench benchmark. If $f(a_1) < f(b_1)$, the minimum lies in the range $[a_0, b_1]$, otherwise it lies in the range $[a_1, b_0]$. Where f is the parameter discovery function. A special constant, ρ , is used to divide the x axis into two segments, long and short. The Golden Section is the equation: $\rho/(1-\rho) = (1-\rho)/1$, i.e. ratio of short to long is equal to ratio of long to the sum of the two. The search was made more efficient by ensuring that the hackbench benchmark only runs once per iteration.

5.7 Results

Partial results for meta-optimization of the PSO algorithm are given in table 5.1 and described in subsection 5.6.1 of Section 5.6. The rsum value is used to compare the various runs (a combination of the minimum response achieved and the number of iterations required to achieve this). The minimum response is more important, by the summation. We fitted a second order model to these results and accepted the model because of high significance of the intercept, ω and ϕ_g . From our results, the value of ϕ_p , the cognitive parameter did not have any effect on the speed of convergence or minimum response achieved of the PSO algorithm. The optimal parameter values were then obtained by the use of the rsm package's *canonical path* optimization function. This makes use of gradients to find the steepest ascents in two directions. The values are: $\omega = 0.4365$ and $\phi_g = 3.020$. For completeness, ϕ_r was also set to 3.020.

The parameters for the PSO algorithm were obtained using the experiments displayed in table 5.1. There are two results for each experiment that are important. One is R^* , the minimum turnaround time, and the other is I^* , the minimum number of iterations (i.e. hackbench executions) needed to achieve the minimum turnaround time. The rsum value is an addition of R^* and I^* and is needed in order to compare the various runs of the experiment. The PSO with default parameters of $\omega = 0.9$, $\phi_p = 4$, and $\phi_g = 4$ are compared with the optimized parameters using a workload on LinSched consisting of 5 groups of 40 processes each and 1 iteration of sending of messages. Both the default and optimized PSO versions yield a turnaround time of 3028 Jiffies, but the optimized one converges faster, requiring just 61 executions of the benchmark, while the default one requires 78 executions.

We then ran the PSO algorithm using a hackbench workload on the LinSched simulator consisting of 5 groups of 40 processes each and 1 iteration of sending of messages. The initial positions for the particles were obtained using a uniform random number generator based on the maximum and minimum limits for the three scheduler tuning parameters (see Section 5.6 for these limits). Our initial results are shown in figures 5.2 and 5.3. Our optimized PSO produced the smallest turnaround time for the benchmark, 3028 Jiffies, while Golden Section algorithm 1 and 2 both produced Jiffies of 5939. However, it took Golden Section algorithm 1 just 8 iterations (benchmark executions) to reach its minimum, and Golden Section algorithm 2 9 iterations to reach

its minimum, while it took the optimized PSO algorithm 31 benchmark executions to reach a minimum of 5939, equivalent to the Golden Section algorithms. From these results, it is clear that the optimized PSO algorithm yields the smallest turnaround time for our benchmark, but the Golden Section algorithms can yield useful results in few iterations. More results are shown in figures 5.4 and 5.5, this time making use of a wider range of workloads, which were derived by increasing the number of sets of messages sent from each sender to each receiver in the benchmark from 1 to 9.

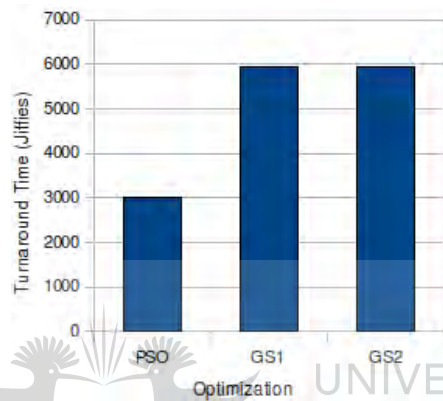


Figure 5.2: Minimum turnaround times achieved

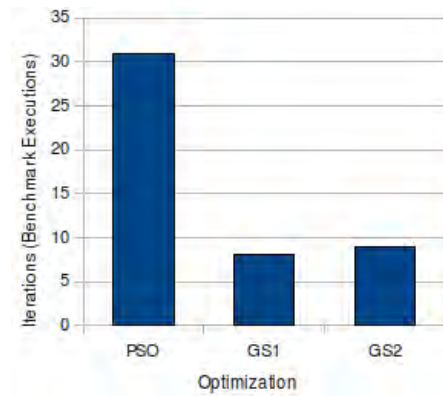


Figure 5.3: Minimum iterations needed to achieve target turnaround

5.8 Standard Optimization-Based Parameter Tuning Methodology

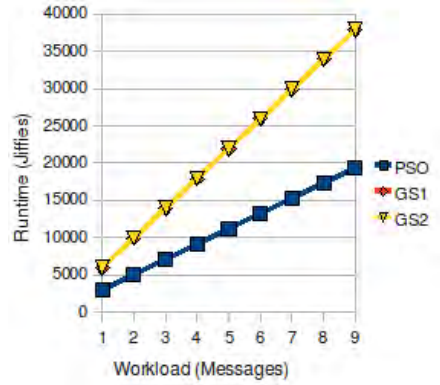


Figure 5.4: Minimum turnaround times achieved for a range of workloads

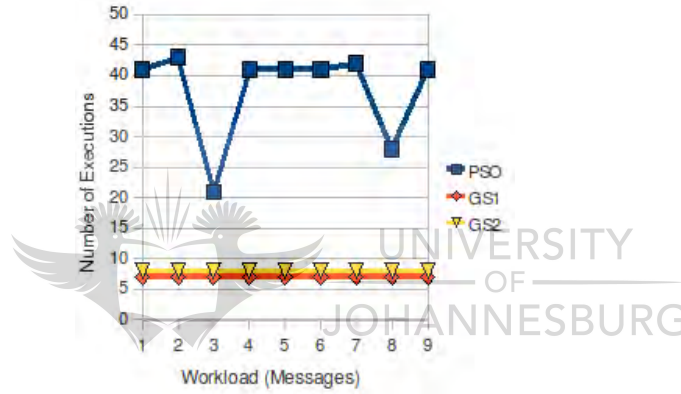


Figure 5.5: Minimum iterations needed to achieve target turnaround, for a range of workloads

5.8 Standard Optimization-Based Parameter Tuning Methodology

Algorithm 5.4 shows a possible methodology for tuning the operating system scheduler.

Algorithm 5.4 Standard optimization-based operating system scheduler tuning methodology

- 1: Get workload
 - 2: Get scheduler tuning parameter ranges
 - 3: Use PSO or Golden Section to explore parameter space with experiments
 - 4: Get optimal parameter values
 - 5: Run workload in production environment with optimal parameter values
-

5.9 Conclusion

This chapter has made several contributions. It has applied standard optimization methods to the task of operating system scheduler tuning. It has demonstrated the usefulness of Particle Swarm Optimization and used meta-optimization to come up with optimal parameters for the algorithm. It has also demonstrated how operating system kernel source code can be inspected in order to convert an optimization problem from one consisting of multiple design variables to one consisting of just one design variable. That is *SchedPeriod* instead of *Latency*, *MinGran*, and *WakeupGran*. This conversion then makes it possible to make use of a local optimization approach, the Golden Section method. The optimized PSO algorithm yields the smallest turnaround time for our workload (the hackbench benchmark), but Golden Section yields useful results at a fraction of the time taken by PSO; 8 benchmark executions as opposed to 31. The next chapter looks at another operating system scheduling problem: Processor Allocation. This is important in order for optimal assignment of CPU cores to tasks in a multicore environment.



Chapter 6

Black Box Learned Optimization Scheduling

6.1 Summary

In the previous two chapters we reported on studies which solve the operating system scheduler tuning problem. In this chapter we report on studies which solve another operating system scheduling problem, Processor Allocation, which has to do with which cores are assigned to which tasks in a multicore environment. Scheduling for multicore computer systems is a challenge since subsets of cores may share resources, such as a cache. Performance for workloads may therefore vary depending on which tasks are scheduled to run on the same subset of cores. There is therefore a need for contention-aware scheduling. Our study involves implementation of an optimal multicore scheduler which has perfect prediction of negative consequences of all combinations of tasks scheduled on the hardware platform we are using. It uses an Integer Program. We show that it is indeed optimal for our workloads, supporting its authors' claims [20]. We also implemented a scheduler which uses a combination of a machine learning model (M5 Prime) and Linear Programming to schedule tasks on CPU cores and compared it to a state-of-the-art scheduler known as Distributed Intensity (DI). Some workloads exhibited moderate improvements in unfairness, but not much in runtime. A scheduler based on another machine learning model was implemented, this time a Multilayer Perceptron (MLP). To do this, we had to generate workloads for the Optimal Scheduler, store its decision for various types of workloads, and train a Multilayer Perceptron model on this

data. We then implemented a scheduler with the Multilayer Perceptron model, which tries to mimic the decision of the Optimal Scheduler. Our results show that our scheduler is better than DI in 7 out of 9 test workloads (mostly by 10%), and approximately equal to the Optimal Scheduler in 6 out of 9 test workloads (exactly equal in 4). The MLP scheduler is faster than the Optimal Scheduler by a wide margin, and faster than the Linear Programming scheduler. We also implemented a Support Vector Machine (SVM) scheduler, similar in design to the MLP scheduler. This scheduler is approximately equal to the Optimal Scheduler in 7 out of 9 workloads (exactly equal in 5). We selected MLP and SVM machine learning models because they are both advanced universal approximators [88, 89]. They are capable of learning complex relationships between independent and dependent variables. The MLP is the most popular type of Neural Network [89, 90].

The rest of this chapter is organized as follows: Section 6.2 has an overview of the problem being addressed, Section 6.3 discusses related work on scheduling, Section 6.4 discusses the Optimal Scheduling algorithm and Distributed Intensity (DI), Section 6.5 discusses our implementation environment, Section 6.6 describes our Linear Programming-based Scheduler (ModLP), Section 6.7 introduces Learned Optimization, Section 6.8 describes the MLP and SVM machine learning models and algorithms, Section 6.9 describes the MLP and SVM schedulers, Section 6.10 presents the results of our experiments, including a discussion on our workloads, and on the results themselves, and Section 6.11 concludes.

6.2 Processor Allocation Introduction

The Operating System scheduler assigns tasks to Central Processing Units (CPUs) [2, 3, 4]. The scheduler decides which tasks will use a particular CPU next and for how long. Modern CPU designs have shifted from trying to increase clock speeds of CPUs to incorporating more processor cores on a single CPU chip, providing facilities for parallel execution of programs, similar to the traditional SMP (Symmetric Multiprocessing) architectures [91, 92, 93]. Instead of having multiple CPUs, modern computers have multiple cores on a single CPU, which is cheaper and more energy efficient. With the advent of multicore systems, the processing elements (cores) share some resources, for example the LLC (last level cache), memory controller, memory bus, and prefetch

hardware. Therefore when two or more tasks are running on cores that share these resources, there is resource contention; their performance will be degraded, compared to if each task were to run on its own [11, 12]. Generally, on multicore architectures, competing tasks face performance degradation when co-scheduled together, on separate cores sharing resources, while cooperating (communicating) tasks face performance degradation when not co-scheduled [13]. Because of this degradation, it is important that when the scheduler is making its decisions, which task should run on which core, it takes resource contention into account; there is a need for *contention-aware schedulers*.

Contention-aware scheduling studies are of two types: those targeted at on-line scheduling, where scheduling decisions must be made while tasks are running, so the scheduler must be very fast and not complicated; and optimal scheduling, where the scheduler is not used on-line, but tries to come up with optimal schedules for the purpose of knowing the best possible schedule, in order to help develop scheduling algorithms [16]. There have been just a handful of studies carried out into optimal contention-aware scheduling; optimal algorithms have been produced as well as heuristic approximations to the optimal algorithms [16, 17, 18, 19, 20]. The problem with optimal algorithms is that they are clairvoyant in that they know in advance the performance degradation of tasks when running in all possible combinations on cores. They are also relatively slow, since they make use of relatively slow optimization methods such as Integer Programming. We achieve performance improvements over the Optimal Scheduler, while generating almost identical schedules.

Also, the heuristic algorithms do not produce schedules as good as the optimal algorithms. The current state-of-the-art contention-aware scheduler is the Distributed Intensity (DI) scheduler [11]. It has the best performance (execution time spent by the scheduler) when generating schedules, however the schedules it produces are not very close to the optimal schedules (most about 10% worse, with our benchmarks). *We combine the on-line and optimal scheduling approaches by developing a scheduler which targets generating optimal schedules while exhibiting fast execution time.* We come up with a method named Learned Optimization, where a model is trained to learn how an optimization algorithm produces optimal schedules. The sub-method we focus on is called Black Box Learned Optimization, because we only work with inputs and outputs of the optimization process. To the best of our knowledge, this approach has not been used before for contention-aware schedulers. We close the gap between

optimal schedulers and approximations, by implementing and evaluating optimization and machine learning-based schedulers.

In our study, we implement an optimal contention-aware scheduling solution [20] which we try to equal using less time-intensive processing. The optimal solution uses Integer Programming. This approach assumes knowledge of various performance values for various combinations of core assignments. It serves as a benchmark. We also implement the DI scheduler, as presented in [11], and our own approach, which combines the DI approach with Linear Programming. We implement predictive models based on machine learning which, given a set of tasks running on a chip, can predict the degradation in task performance for each task. The results of this hybrid approach of combining DI with Linear Programming are not so good, compared to DI. This is mainly because there is an additional level of estimation, first by DI, then with the machine learning models, after which optimization is done. We then implemented a scheduler based on another machine learning model, this time a Multilayer Perceptron [88, 89, 94, 95, 96]. To do this, we had to generate workloads for the Optimal Scheduler. Once the Optimal Scheduler did its work, the Multilayer Perceptron was used to learn, for each type of workload, the assignment of tasks to cores. Generating workloads involved analysing the metrics associated with the workloads available in order to generate new performance figures for new workloads. We then implemented a scheduler with the Multilayer Perceptron model, named MLP, which tries to mimic the decision of the Optimal Scheduler. Our results are very promising. The MLP scheduler is better than DI in 7 out of 9 test workloads, and approximately equal to the Optimal Scheduler in 6 out of 9 test workloads. The MLP scheduler produces schedules very close to the optimal scheduler and much better schedules than DI. Also, it has much faster execution time than the optimal scheduler, and relatively close scheduler execution time to DI. We use the same approach to develop a Support Vector Machine (SVM) scheduler, which produces schedules approximately equal to the optimal in 7 out of 9 workloads tested and exactly equal in 4 out of 9 workloads tested.

6.3 Related Work Summary

While Chapter 2 has details of related work, we have a short summary in this section.

The closest work to ours is that done by Blagodurov et al. [11]. In their study, the authors investigated the best approaches to classifying threads, based on performance metrics obtained via CPU counters on modern Chip Multiprocessors (CMPs). These include metrics such as Cycles Per Instruction (CPI), Instructions Per Cycle (IPC), and Misses Per Instruction (MPI), used to estimate number of cache misses. The classification approaches are used to classify threads when they compete for resources. These approaches are explained in Section 2.4. Based on the classification study, the authors developed a scheduling algorithm known as Distributed Intensity (DI). Their results showed that its performance was close to the Integer Programming-based optimal solution. In our work, we investigated how task degradation could be predicted using machine learning approaches. We selected the best model for use in a scheduler that combines DI with linear programming, and also the best model to generate workloads to train a Multilayer Perceptron Scheduler.

Rai et al. [41, 42] carried out studies on characterizing and predicting L2 cache behavior using various independent variables. They used machine learning algorithms to build models which could then be used for characterizing and predicting cache behavior.

The machine learning algorithms they used were Linear Regression, Artificial Neural Networks, Locally Weighted Linear Regression, Model Trees, and Support Vector Machines. The class variable to be predicted was named “solo run L2 cache stress”. The attributes (independent variables) used were

- i. L2 cache references per kilo instructions retired.
- ii. L2 cache lines brought in (due to miss and prefetch) per kilo instructions retired.
This shows the stress put by a program on the L2 cache.
- iii. L2 cache lines brought in (due to miss and prefetch) per kilo L2 cache lines referenced. This shows the re-referencing tendency of a program.
- iv. Fractional L2 cache occupancy of a program. This gives a rough estimate of fraction of space occupied by a program in the L2 cache while sharing with another program.

Rai et al. [43] extended their work by developing a machine learning based meta-scheduler. They used their predictive models mentioned in the previous paragraph to aid scheduling decisions, achieving a 12% speedup over the Linux CFS scheduler.

Our work combines machine learning not just to predict degradation in performance, but to assign tasks to cores. We base our solutions on the optimal solution, and achieve schedules as good as, or very close to, the optimal solution. We do not use as many statistics for prediction as Rai et al., making our method compatible with a wider range of CPU architectures.

Illikkal et al. [36] developed a rate-based approach to resource management of CMPs. They make use of rate-based Quality of Service (QoS) methods to decrease resource contention. They make use of hardware Voltage and Frequency (V/f) Scaling and Clock Modulation/Gating. V/f involves changing the frequency of a core in order to reduce power consumption, while Clock Modulation involves feeding the clock to the processor for short durations. Illikkal et al.'s results showed their approach to be flexible and effective at ensuring QoS. Our approach does not require hardware manipulation but simply reading of certain CPU counters.

Hoh [37] made use of a vector-based approach to co-schedule tasks which were dissimilar in their use of CPU resources. Each task had a vector associated with it, which included elements for various CPU performance counters, such as number of all instructions completed, number of floating point instructions, number of special (e.g. graphic) instructions, number of L2 cache cycles, etc. When selecting a task, the task with the greatest vector difference from the average is selected. Our approach uses just one metric, so is more simple and faster.

Klug et al. [38] implemented a scheduling system which forces pinning of tasks to specific CPU cores. After a time constant, their autopin system changes the pinning so various pinning combinations are tried. Once the best pinning combination is discovered, the tasks use it until they terminate. Autopin can be viewed as a brute-force approach to processor allocation, while our method is more intelligent.

Our work tries to make use of machine learning and optimization to develop an innovative and efficient solution for multicore scheduling, equal to or very close to the optimal in quality, and succeeds. Our approach is more practical than the optimal because it is faster.

6.4 The Optimal Scheduling Algorithm and a Relatively Near-Optimal Scheduling Algorithm

6.4.1 Optimal Scheduling

Jiang et al. [20] carried out a study on optimal scheduling of tasks on a multicore system. Their optimal solution is based on Integer Programming. It solves a partition problem, in which n jobs are in $m = \frac{n}{u}$ sets, where m is the number of chips, each having u cores. Each set is as large as the number of cores in a chip. The objective function, as given by Jiang et al. [20], is:

$$\min \sum_{i=1}^{\binom{n}{u}} d(S_i) \cdot x_{S_i} \quad (6.1)$$

The constraints are:

$$x_{S_i} \in \{0, 1\}, \quad 1 \leq i \leq \binom{n}{u};$$

$$\sum_{k:1 \in S_k} x_{S_k} = 1; \sum_{k:2 \in S_k} x_{S_k} = 1; \dots; \sum_{k:n \in S_k} x_{S_k} = 1.$$

x_{S_i} is a binary variable, 1 if S_i is one of the sets in the final partition result or 0 if it is not. $|S_i| = u$.

$$d(S_i) = \sum_{j \in S_i} d_{j, S_i - \{j\}} \quad (6.2)$$

$d(S_i)$ is the sum of degradations of all the tasks in S_i , when they co-run on a single chip, as opposed to running alone on the same chip. Co-run degradations are obtained by measuring the CPI (Cycles Per Instruction) when a task runs alone, and when it runs together with the other tasks in its set. We use the inverse (Instructions Per Cycle) instead. This Integer Programming formulation of the multicore scheduling problem has been shown to be optimal [20].

6.4.2 Relatively Near-Optimal Scheduling

The algorithm known as DI (Distributed Intensity) gives results relatively close to the Optimal Scheduler, with very low runtime costs [11]. It uses a particular metric for tasks: MPI (Last Level Cache Misses Per Instruction). These are measured when a task runs on its own, with no other tasks running on cores on the same chip. Performance degradation factors made worse by a high miss rate [15]. A high miss rate means more

evictions from the cache and therefore more cache contention, the prefetching hardware is triggered more often leading to higher prefetching hardware contention, and more memory bus traffic and more requests to the memory controller, which in means higher memory bus and memory controller contention.

Tasks with high MPI are cache intensive, so the idea is to schedule them on chips with cores running other non-cache intensive tasks; basically, to distribute the load. This is done by sorting the tasks to be scheduled in a list based on MPI, then picking the first and last to run on a chip, the next first and next last to run on another chip, etc. In our study, we implement DI and try to beat its performance, and actually succeed.

6.5 Implementation Environment

We carried out all our experiments in an operating system scheduling simulator known as AKULA [12, 19]. AKULA presents various classes in an object-oriented framework which are used to implement custom schedulers and to modify the simulation environment. It has two ways of running simulations. One is bootstrapping, which involves logging performance of benchmarks running in various combinations on a multicore platform. For example, if the architecture being used comprises two sets of four cores each, with the 4 cores sharing resources such as the Last Level Cache (LLC), and there are eight benchmarks, then there are 8 ways of running the workload one benchmark at a time on a set of 4 cores, there are 28 ways of running 2 benchmarks on a set of 4 cores, there are 56 ways of running 3 benchmarks on the set of 4 cores, and there are 70 ways of running 4 benchmarks on a set of 4 cores. All these combinations must be evaluated by running the actual benchmarks and logging performance, using a tool in AKULA called the profiler.

Once the profiler has done its work and a scheduling algorithm has been implemented in AKULA, the scheduler can be tested using bootstrapping, which will keep track of the progress each task makes per clock tick, depending on which other tasks are running with it in the same set of cores sharing critical resources, based on performance data obtained by the profiler. When a task has made up to 100% progress, it terminates. This means that the task has run for enough ticks to simulate its required

execution time; the simulator will thus change the status of the task to completed and remove it from the set of tasks still competing for CPU time.

Another way of running a simulation in AKULA is to attach benchmarks tasks to actual hardware cores using an appropriate utility. Thus, the scheduler runs in user space on an operating system, but is still able to control placement of tasks on cores. In our work, we make use of the bootstrapping approach to evaluating our scheduling algorithms. The architecture we used for our experiments consists of two sets of four cores each, with each set sharing the Last Level Cache.

6.6 A Linear Programming-Based Multicore Scheduler

In order to develop a Linear Programming-based multicore scheduler, we need to have a model which, given some performance figures of tasks, can predict the degradation in performance (a sort of cost function). We experimented with several prediction models, including Multilayer Perceptron, Linear Regression, REPTree, and M5 prime. The Multilayer Perceptron had a correlation coefficient of 0.7616 (1 would show perfect correlation between predicted output and actual output). The Linear Regression model had a correlation coefficient of 0.8332, REPTree had a correlation coefficient of 0.99 and the M5 Prime model had a correlation coefficient of 0.9906. Since the M5 Prime had the best prediction results, it was chosen to implement our own scheduling algorithm, based on Linear Programming. We focus on a specific case of a two-chip platform, with each chip having four cores, giving eight cores in total. The problem is formulated as an Assignment Problem [97, 98], one of assigning tasks to agents (in this case, tasks to cores). Figure 6.1 illustrates the assignment problem.

The optimization problem is formulated as follows:

$$\text{Min} \quad \sum_{i=1}^m \sum_{j=1}^n d(x_{ij})$$

Subject to

$$\begin{aligned} \sum_{i=1}^m x_{ij} &= 1, \quad \forall j = 1, 2, \dots, n \\ \sum_{j=1}^n x_{ij} &= 4, \quad \forall i = 1, 2, \dots, m \end{aligned}$$

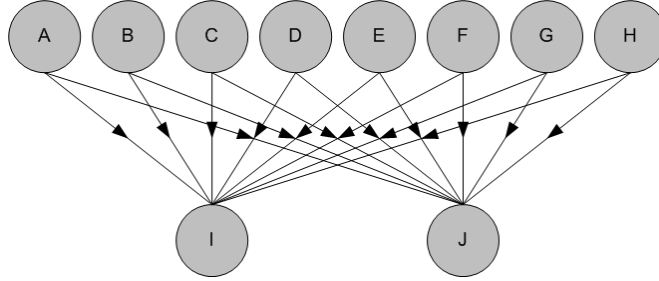


Figure 6.1: An Assignment Problem Graph: Vertices A-H are the tasks, each of which has to be assigned to one of two sets of cores, I and J. Each edge has a weight, which is the cost of assignment to a particular set of cores. In the model used in this thesis, the cost is also dependent on other assignments.

x_{ij} is 1 when task j is scheduled on core i , and is 0 otherwise. $d(x_{ij})$ is the degradation of the task when run in its “home” set. This degradation is a prediction obtained using an M5 Prime model. The first constraint states that the sum of all x_{ij} for a particular j is 1, meaning that any given task is only assigned to one set of cores (there are two sets of 4 cores each). The second constraint states that the sum of all x_{ij} for a particular i is 4, meaning that a set of 4 cores is assigned exactly 4 tasks.

The home set is obtained by sorting the 8 tasks in the workload according to their MPI (cache Misses Per Instruction) metrics. This method was used by Blagodurov et al. in [11] in the DI scheduling algorithm. The reasoning is that tasks with high MPI should be scheduled with tasks with low MPI, since the high MPI tasks are cache-hungry and will get in each others way if scheduled together. When the 8 tasks in the workload are sorted by their MPIs, the 1st and the last can be removed from the sorted set and assigned to cores, until there are no tasks left in the sorted set. This means that the sum of all MPIs is distributed evenly among the chips. Our scheduling system thus combines DI with Linear Programming. This hybrid approach is meant to improve on the results of DI.

6.7 Learned Optimization

In our work, we came up with the term Learned Optimization. Learned Optimization is a process of carrying out optimization using Machine Learning (ML) models and algorithms, for scheduling purposes. Optimization problems are formulated and solved using some optimization algorithm, for example Linear or Integer Programming.

Machine Learning algorithms then learn how the optimization is done. Optimization algorithms can be viewed as black boxes which map inputs to one or more outputs. In our case, task attributes are mapped to scheduling decisions by an optimization algorithm, for example Integer Programming. We can accumulate many such mappings and use that to train an ML model, which would then mimic the behavior of the original optimization algorithm, when presented with new, previously unseen data. We call this variant Black Box Learned Optimization. This is similar to “System Identification” used in control engineering to build mathematical models of dynamical systems [99, 100]. Generalization of machine learning models is described in various publications, such as Bishop [89] and Ayodele [90]. We do not build mathematical models but store machine learning models for use at a later point in time by schedulers. Algorithm 6.1 describes general Black Box Learned Optimization.

Algorithm 6.1 General Black Box Learned Optimization Algorithm. Instances input into the optimization algorithm are independent variable values.

```
1: while Not EndOfInstances do
2:   Load Next Instance
3:   Run Optimization Algorithm on Instance
4:   Save Result
5: end while
6: Load Instance and Result Pairs
7: Train Machine Learning Model with Instance and Result Pairs
8: Use Trained Model (e.g. in a scheduler)
```

6.8 MLP and SVM

This section gives an overview of the MLP (Multilayer Perceptron) and SVM (Support Vector Machine) machine learning models.

6.8.1 MLP

Learning occurs in the brain in animals [88]. Scientists have been trying to understand how it works so that the process can be copied and used in machine learning systems. The processing units of the brain are nerve cells called neurons. There are approximately 100 billion neurons in the human brain. If the potential of a neuron’s membrane

reaches some threshold, the neuron fires and a pulse of fixed strength and duration is sent to the axon. Transmitter chemicals within the brain cause the electrical potential within the neuron to vary. Axons divide into connections to many other neurons and connect to each of these neurons in a synapse. Each neuron is connected to thousands of other neurons. There are approximately 100 trillion synapses in the human brain. Learning occurs when the strength of connections between neurons is modified, and also when new connections are created.

McCulloch and Pitts neurons are mathematical models of neurons. They include:

- i. A set of weighted inputs w_i that correspond to the synapses.
- ii. An adder that sums the input signals (equivalent to the membrane of the cell that collects electrical charge).
- iii. An activation function (a threshold function) that decides whether the neuron fires for the given inputs.

McCulloch and Pitts neurons have limited functionality but networks of them can carry out tasks like learning functions and memorizing pictures.

The Perceptron is a collection of McCulloch and Pitts neurons together with a set of inputs and some weights on links from inputs to the neurons. Connecting several neurons in a system results in neural networks. The basic neural network model can be described as a series of functional transformation [89]. First M linear combinations of the input variables x_1, \dots, x_D are constructed as

$$a_j = \sum_{i=1}^D w_{ji}^{(1)} x_i + w_{j0}^{(1)} \quad (6.3)$$

where $j = 1, \dots, M$ and the superscript (1) indicates that the corresponding parameters are in the first “layer” of the network. $w_{ji}^{(1)}$ are weights and $w_{j0}^{(1)}$ are biases. The quantities a_j are known as activations. Each activation is transformed using a differentiable, non-linear activation function $h(\cdot)$ to give

$$z_j = h(a_j) \quad (6.4)$$

These quantities correspond to the outputs of hidden units. The non-linear functions $h(\cdot)$ are usually chosen to be sigmoidal functions. These values are again linearly

combined to give output activation functions.

$$a_k = \sum_{j=1}^M w_{kj}^{(2)} z_j + w_{k0}^{(2)} \quad (6.5)$$

where $k = 1, \dots, K$, K being the total number of outputs. This is the second layer of the network. The output unit activations are transformed using an appropriate activation function to give a set of network outputs y_k . If the neural network is to be used for regression problems, $y_k = a_k$. For binary classification problems, each output unit activation is transformed using a logistic sigmoid function, so

$$y_k = \sigma(a_k) \quad (6.6)$$

where the sigmoid function,

$$\sigma(a) = \frac{1}{1 + \exp(-a)} \quad (6.7)$$

These stages can be combined to give the overall network function that takes the form

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma \left(\sum_{j=1}^M w_{kj}^{(2)} h \left(\sum_{i=1}^D w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right) \quad (6.8)$$

where all weights and biases are grouped into vector \mathbf{w} . The Multilayer Perceptron is a non-linear function from inputs to outputs, controlled by the weight parameters. Figure 6.2 shows an MLP model, as described in this section. The MLP is the most popular type of Neural Network [89, 90].

Training the MLP consists of two parts: working out what the outputs are for given inputs and weights, and then adjusting the weights according to the prediction error between the actual output and target output. The N weights and thresholds, together with the error, form an error surface, when plotted in $N + 1$ dimensions. Network training tries to find the lowest point in this many-dimensional surface. From an initial random configuration, the training algorithms seek for the global minimum in an incremental manner. Typically, the gradient of the error surface is calculated at the current point and a move in a downhill direction is made.

Back Propagation is the best known example of a neural network training algorithm [90]. In this algorithm, the gradient vector of points on the error surface are calculated. Moves are made in the direction of steepest descent. It is difficult to know how large

the steps should be. Small steps are more precise but slower, while large steps are faster but may go off in the wrong direction or overstep the solution. The iterations of MLP training are called epochs. Training stops when a given number of epochs is reached, or when the error level is acceptable, or when there is no improvement in error of a certain number of epochs.

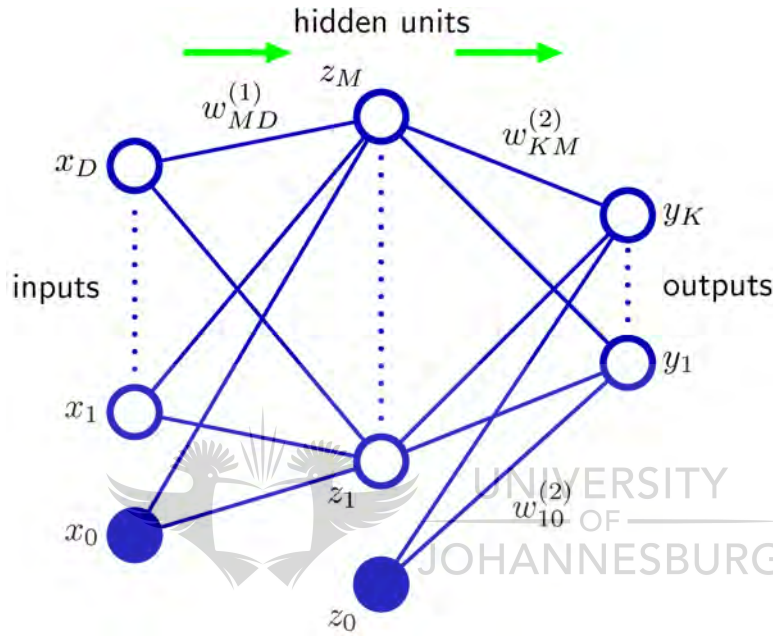


Figure 6.2: Multilayer Perceptron. The input, hidden, and output variables are represented by nodes, and the weights are represented by links between nodes. x_0 and z_0 are bias nodes. The arrows show the direction of information flow through the network.

6.8.2 SVM

An SVM (Support Vector Machine) is a machine learning model with its associated learning algorithms, introduced by Vapnik in 1992 and developed further in later years [101]. SVM implements the idea of building a model of a line which separates data belonging to two classes, optimally. However, it may not be possible to come up with such a straight line; therefore SVM also transforms data into higher dimensions, such that a plane (if three-dimensional) or hyperplane (if higher dimensions are used) can separate the data. This is generally what *kernel methods* do [88]. The kernels are the functions which transform the data for convenient classification. Figure 6.3 shows the power of kernel methods in aiding classification.

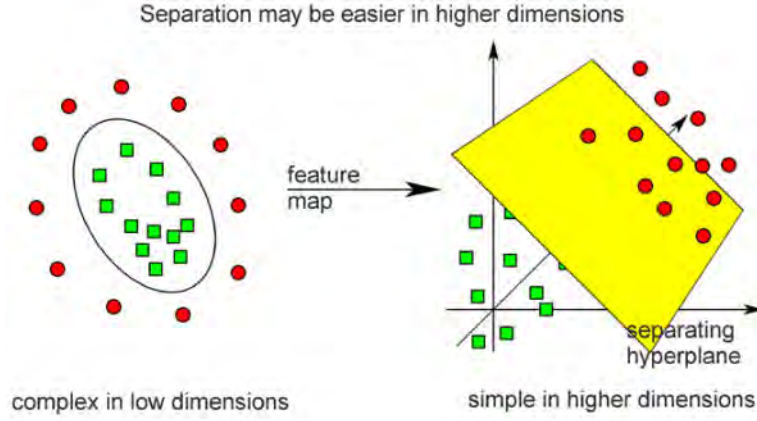


Figure 6.3: Transformation of data to aid in separation [1].

A common formulation of SVM problems follows [102]. This is the implementation in the LibSVM software library used in our work. Given training vectors $\mathbf{x}_i \in R^n, i = 1, \dots, l$, in two classes, and a vector of classes $\mathbf{y} \in R^l$ such that $y_i \in \{1, -1\}$, the C-SVC (C-Support Vector Classification) solves the following optimization problem.

$$\begin{aligned} \text{Min}_{\mathbf{w}, b, \xi} \quad & \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^l \xi_i \end{aligned} \quad (6.9)$$

$$\text{Subject to} \quad y_i (\mathbf{w}^T \phi(\mathbf{x}_i) + b) \geq 1 - \xi_i, \quad (6.10)$$

$$\xi_i \geq 0, \quad i = 1, \dots, l, \quad (6.11)$$

where $\phi(\mathbf{x}_i)$ maps \mathbf{x}_i into a higher-dimensional space. $C > 0$ is a regularization parameter. The corresponding dual problem, which is necessary because of possible high dimensionality of the vector \mathbf{w} , is as follows.

$$\text{Min}_{\alpha} \quad \frac{1}{2} \alpha^T Q \alpha - \mathbf{e}^T \alpha \quad (6.12)$$

$$\text{Subject to} \quad \mathbf{y}^T \alpha = 0, \quad (6.13)$$

$$0 \leq \alpha_i \leq C, \quad i = 1, \dots, l, \quad (6.14)$$

where $\mathbf{e} = [1, \dots, 1]^T$ is the vector of all ones, Q is an l by l positive semidefinite matrix, $Q_{ij} \equiv y_i y_j K(\mathbf{x}_i \mathbf{x}_j)$, and $K(\mathbf{x}_i \mathbf{x}_j) \equiv \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$ is the kernel function.

After the dual problem is solved, using the primal-dual relationship, the optimal w

satisfies

$$\mathbf{w} = \sum_{i=1}^l y_i \alpha_i \phi(\mathbf{x}_i) \quad (6.15)$$

and the decision function is

$$\text{sgn}(\mathbf{w}^T \phi(\mathbf{x}) + b) = \text{sgn} \left(\sum_{i=1}^l y_i \alpha_i K(\mathbf{x}_i, \mathbf{x}) + b \right) \quad (6.16)$$

The SVM model stores $y_i \alpha_i \quad \forall i, b$, label names, support vectors, and kernel parameters in the model for prediction.

For multi-class classification (more than two classes), a “one-against-one” approach is used. Given k classes, $k(k-1)/2$ classifiers are constructed and each one trains data from two classes. For training data from the i^{th} and j^{th} classes, the following two-class classification problem is solved.

$$\begin{array}{ll} \text{Min}_{\mathbf{w}^{ij}, b^{ij}, \xi^{ij}} & \frac{1}{2} (\mathbf{w}^{ij})^T \mathbf{w}^{ij} + C \sum_t (\xi^{ij})_t \end{array} \quad (6.17)$$

$$\text{Subject to} \quad (\mathbf{w}^{ij})^T \phi(\mathbf{x}_t) + b^{ij} \geq 1 - \xi_t^{ij}, \text{ if } \mathbf{x}_t \text{ in the } i^{\text{th}} \text{ class} \quad (6.18)$$

$$(\mathbf{w}^{ij})^T \phi(\mathbf{x}_t) + b^{ij} \leq -1 + \xi_t^{ij}, \text{ if } \mathbf{x}_t \text{ in the } j^{\text{th}} \text{ class} \quad (6.19)$$

$$\xi_t^{ij} \geq 0. \quad (6.20)$$

A voting strategy is used: each two-class (binary) classification is considered to be a voting where votes are cast for all data points \mathbf{x} . A point is designated to be in the class with maximum number of votes.

Various kernels are available in SVMs. LibSVM has the following kernels [103].

- i. Linear. $K(x_i, x_j) = x_i^T x_j$.
- ii. Polynomial. $K(x_i, x_j) = (\gamma x_i^T x_j + r)^d, \gamma > 0$.
- iii. Radial Basis Function (RBF). $K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2), \gamma > 0$.
- iv. Sigmoid. $K(x_i, x_j) = \tanh(\gamma x_i^T x_j + r)$.

γ, r , and d are kernel parameters.

6.9 MLP and SVM-Based Multicore Schedulers

This thesis builds on an idea from a study carried out by Kirschner on learning from optimization models [104]. In his study, he used machine learning algorithms to learn how optimization methods work, in order to generate rules and models that could then be used to carry out scheduling for processing of chemicals, without the long time delays when regular optimization methods are used. Our work differs in the nature of the problem, multicore scheduling as opposed to chemical process scheduling, the optimization method to be learned from (we use Integer Programming), and also in the Machine Learning tools used; we use M5 Prime, Multilayer Perceptron, and SVM models while Kirschner used Decision Trees, Nearest Neighbour, Bagging, Boosting, and Winnow. We used M5 Prime to generate artificial workloads to train the Multilayer Perceptron model. Algorithm 6.2 shows the operation of the main simulator module, algorithm 6.3 shows the operation of the bootstrap module, and algorithm 6.4 shows the operation of the scheduler module. The sequence of steps used to develop the Multilayer Perceptron and SVM multicore schedulers is shown in algorithm 6.5.

Algorithm 6.2 Simulation main module

```
1: Load machine learning Model
2: Load machine model
3: while Not EndOfFile do
4:   Read workload data from file
5: end while
6: while More workloads to process do
7:   Execute bootstrapping with {ML model, workload, machine model}, invoking
     Algorithm 6.3.
8: end while
9: Save statistics
10: Exit
```

Algorithm 6.3 Simulation bootstrap module

```

1: Create scheduler object
2: while Not all tasks finished do
3:   Wallclock++
4:   Calculate progress of all tasks
5:   Check for completion
6:   Call scheduler to update schedule and statistics, invoking Algorithm 6.4
7: end while

```

Algorithm 6.4 Simulation scheduler creation module

```

1: Create task array for workload
2: Sort task array
3: Use ML model to assign tasks to cores

```

6.10 Results

6.10.1 Workloads

The benchmark statistics in the workloads were obtained from the AKULA software release [12]. The actual benchmarks are from the SPEC CPU2006 benchmark suite [59]. We selected 9 benchmarks, based on the availability of performance figures in AKULA, and also since they represent a good mix of shared-resource-intensive and shared-resource-non-intensive applications [15]. They are: *gcc*, *mcf*, *gobmk*, *libquantum*, *gamess*, *milc*, *namd*, *povray*, and *lbm* and are described in Table 6.1. The 9 benchmarks were combined together in 9 different ways, giving the 9 workloads which we used for our experiments (see Table 6.2). Each workload has 8 benchmarks. For the purpose of evaluating performance of computer-bound workloads, the same number of tasks are used as there are cores. In actual compute-bound environments, this is normally the case [11].

6.10.2 Schedulers

We tested the 6 different schedulers described so far in this chapter.

- i. Optimal. Based on Integer Programming model and knowledge of degradations for all combinations of tasks running on sets of 4 cores on our test platform.
- ii. DI. Distributed Intensity scheduler used by Blagodurov et al. [11].

Algorithm 6.5 MLP and SVM Scheduler Development

- 1: Use available workload to generate prediction model based on M5 Prime and MPI metric
- 2: Fit log normal distribution to original figures
- 3: Generate new MPI and runtime length figures for new tasks (i.e. new workloads) using the log normal distribution. We generated 165 workloads with 8 tasks each.
- 4: Run Optimal scheduler on the new workloads and log the various scheduling decisions (assignment of tasks to cores for each workload)
- 5: Train Multilayer Perceptron (MLP) and Support Vector Machine (SVM) on this new core assignment data and save the model. The training data consisted of 10,000 instances; each instance specifies how one task is scheduled when seven other tasks are also competing to be scheduled.
- 6: Implement schedulers in AKULA simulator to make use of the MLP and SVM models to assign tasks to cores for test workload
- 7: Capture various performance metrics

Table 6.1: SPEC CPU2006 benchmark descriptions. Type I is Integer, F is Floating Point.

Benchmark	Type	Description
gcc	I	GNU C Optimizing Compiler benchmark
mcf	I	Single-depot vehicle scheduling in public mass transportation benchmark (Integer)
gobmk	I	Go-playing and analyser benchmark (Integer)
libquantum	I	Quantum computer simulation benchmark (Integer)
gamsess	F	Quantum chemical computation benchmark
milc	F	MIMD Lattice Computation benchmark, simulating lattice gauge theory
namd	F	Benchmark simulating large biomolecular systems
povray	F	Persistence of Vision ray-tracer benchmark
lbm	F	Benchmark simulating incompressible fluids using Lattice Boltzmann Method

- iii. ModLP. DI used as a starting point followed by Linear Programming optimization. It uses an M5 Prime model for prediction.
- iv. MLP1. Multilayer Perceptron with 8 input nodes, 2 output nodes (1 for each chip), and 1 hidden layer with 5 nodes. We experimented with various numbers of hidden layers and hidden layer nodes. A 5-node configuration produced worse scheduling decisions than a 2-node configuration, but was better than the other configurations we experimented with. The MLP1 model had an accuracy of 77.35%, a Precision score of 0.777, a Recall score of 0.773, and an F-Measure

Table 6.2: Workload Composition

#	Constituent Applications
1	gcc, lbm, mcf, milc, povray, gamess, namd, gobmk
2	gcc, lbm, mcf, milc, povray, gamess, namd, libquantum
3	gcc, lbm, mcf, milc, povray, gamess, gobmk, libquantum
4	gcc, lbm, mcf, milc, povray, namd, gobmk, libquantum
5	gcc, lbm, mcf, milc, gamess, namd, gobmk, libquantum
6	gcc, lbm, mcf, povray, gamess, namd, gobmk, libquantum
7	gcc, lbm, milc, povray, gamess, namd, gobmk, libquantum
8	gcc, mcf, milc, povray, gamess, namd, gobmk, libquantum
9	lbm, mcf, milc, povray, gamess, namd, gobmk, libquantum

Table 6.3: Average degradation and unfairness performance figures for various workloads and Optimal, DI, ModLP, MLP1, MLP2, and SVM Schedulers (Best Figure is Bold; Average Degradation not compared with Optimal)

Workload	Avg Degradation Percentage						Unfairness				
	Optimal	DI	ModLP	MLP1	MLP2	SVM	Optimal	DI	ModLP	MLP2	SVM
1	15.44928	15.89507	18.14677	15.89507	15.36548	15.36548	209.85335	195.45571	168.94460	201.54266	201.54266
2	9.06622	10.34547	13.64042	10.34547	9.12833	9.12833	133.95346	132.30041	147.66707	132.92729	132.92729
3	21.65493	24.58019	24.59467	24.58019	21.65493	21.65493	171.26594	168.86807	169.19926	171.26594	171.26594
4	21.72809	24.77222	24.61435	24.77222	21.72809	21.72809	170.77428	167.87780	169.13020	170.77428	170.77428
5	21.62507	24.54359	24.60900	24.54359	21.62507	21.62507	171.64460	169.47427	169.16689	171.64460	171.64460
6	16.53802	18.04719	18.55790	18.04719	16.95042	16.95042	186.27800	189.81524	170.80955	188.04769	188.04769
7	16.22186	16.37744	18.48113	16.37744	18.74328	16.37744	189.71004	190.12294	165.65228	207.81127	190.12294
8	14.37630	14.61350	16.58507	14.61350	14.75016	14.75016	207.00794	205.57945	176.72302	207.48043	207.48043
9	18.19894	20.43654	22.21250	20.43654	18.19894	18.19894	186.29819	187.96851	156.25516	186.29819	186.29819

Table 6.4: Average degradation percentage improvement for DI, ModLP, MLP2, and SVM, for all Workloads

Workload	Avg Degradation Percentage Improvement				
	DI Over Optimal	ModLP Over DI	MLP2 Over DI	MLP2 Over Optimal	SVM Over Optimal
1	-2.89%	-14.17%	3.33%	0.54%	0.54%
2	-14.11%	-31.85%	11.76%	-0.69%	-0.69%
3	-13.51%	-0.06%	11.90%	0.00%	0.00%
4	-14.01%	0.64%	12.29%	0.00%	0.00%
5	-13.50%	-0.27%	11.89%	0.00%	0.00%
6	-9.13%	-2.83%	6.08%	-2.49%	-2.49%
7	-0.96%	-12.85%	-14.45%	-15.54%	-0.96%
8	-1.65%	-13.49%	-0.94%	-2.60%	-2.60%
9	-12.30%	-8.69%	10.95%	0.00%	0.00%

score of 0.773 on the training/test data. The accuracy is average for two classes (chip 1 and chip 2). Precision indicates out of all workloads in the test set classified to run on chip 0, how many were actually scheduled to run on chip 0 by

the optimal scheduler. The same applies to chip 1. Recall indicates out of all workloads scheduled to run on chip 0 by the optimal scheduler, how many were actually classified like that by the model. The F-Measure score is a combination of the two scores, which tries to find a balance between Precision and Recall [105]. The higher the scores the better, generally.

- v. MLP2. Similar to MLP1, but this time the Multilayer Perceptron has only 2 nodes in the hidden layer. A configuration of two nodes in a single hidden layer produced the best results, better than using two or three hidden layers, and a larger number of nodes. The MLP2 model had an accuracy of 67.5%, a Precision score of 0.679, a Recall score of 0.675, and an F-Measure score of 0.673 on the training/test data. The scores for MLP2 are lower than for MLP1 but on the actual workloads, the MLP2 scheduler made better scheduling decisions.
- vi. SVM. The model used had a cost parameter, $C = 8192$ and $\gamma = 0.0001220703125$. The SVM model had an accuracy of 65.9375%, a Precision score of 0.675, a Recall score of 0.659, and an F-Measure score of 0.652.

Table 6.3 shows the results for the various workloads, for the Optimal, DI, ModLP, MLP1 and MLP2 schedulers, for average degradation and unfairness. The better figures are shown in bold. Optimal would have better average degradation for all workloads, hence is in a separate section for that metric. Degradation is given by:

$$Degradation = 100 \times \frac{(runtime - solotime)}{runtime} \quad (6.21)$$

Runtime is the time the task takes to run to completion when scheduled with other tasks, while solotime is the time the task takes to run when scheduled to run alone. Average degradation is the average percentage degradation over all tasks. A lower figure is better. One can see that the MLP and SVM schedulers yield better average degradation for all workloads. MLP2 is better in 7 out of 9 workloads and MLP1 is better in 2 workloads. SVM is better in 8 out of 9 workloads.

For future comparisons, MLP2 is selected as the MLP scheduler of choice, given its impressive performance in the average degradation metric.

When it comes to fairness, ModLP is better than the other 3 schedulers (even Optimal) in 5 out of 9 workloads. Because of how the problem is formulated, it does a

better job of balancing the load. The unfairness metric captures by how much systems improve performance of certain tasks at the expense of other tasks. A lower number is better. Unfairness is calculated as:

$$\text{Unfairness} = 100 \times \frac{\sqrt{\frac{\sum_{i=1}^N d_i^2}{N} - \frac{\sum_{i=1}^N d_i}{N}}}{\frac{\sum_{i=1}^N d_i}{N}} \quad (6.22)$$

where d_i is the degradation of task i and N is the number of tasks. This metric is closely related to the Coefficient of Variation [25]. We want as little variation in degradation among the various tasks as possible. Since the ModLP scheduler first sorts the tasks as in DI, and then balances the load using Linear Programming, utilizing an M5 Prime prediction model to predict the impact of certain scheduling decisions, it has good fairness results.

Table 6.4 gives improvements as percentages; Improvement of Scheduler A over Scheduler B. Bold percentages indicate equal or better performance. Less than one percent difference in the negative direction is also considered equal. One can see that DI is only equal to Optimal in one workload. ModLP is equal to or better than DI in 3 workloads. MLP2 is better than DI in 7 out of 9 workloads and SVM is better than DI in 8 out of 9 workloads. Out of these, 5 are differences of more than 10%. This is a significant improvement over DI. MLP2 is also equal to DI in one workload.

And now, very importantly, MLP2 is equal to Optimal in 6 out of 9 workloads, and strictly equal (0.00%) difference in 4 workloads. SVM is equal to Optimal in 7 out of 9 workloads and strictly equal (0.00%) difference in 4 workloads. This is very significant because it means that Optimal scheduling can be done without going through a time-intensive optimization process such as Integer Programming.

Figure 6.4 shows the average runtimes of the various schedulers evaluated in milliseconds. These are execution times; how long the scheduler spent doing its work per workload, on average. Note that the focus is on the scheduler, not the whole software system which includes the simulator, and which in a live operating system would include other components such as the memory manager. The Optimal scheduler spent the most time per workload: 33.56ms. This is because of its use of Integer Programming. SVM is next, taking 24.22ms. It is faster than the Optimal Scheduler because a more simple SVM classification is used, with a model which has already been trained. ModLP is next, taking 4.41ms per workload. It is faster than the Optimal because

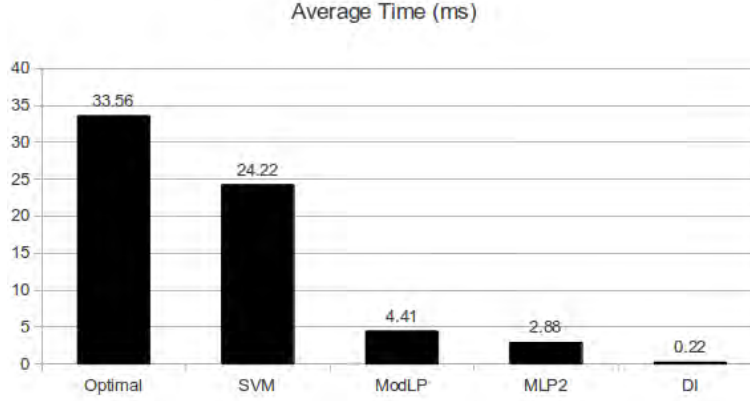


Figure 6.4: Average run times for 5 of the approaches we studied in *ms*.

Linear Programming executes faster than Integer Programming, but it also had to use an M5 Prime model for degradation predictions. MLP2 is next, taking 2.88ms per workload. It is faster than Optimal and ModLP because solving the scheduling problem corresponds to simply plugging in the process MPI figures, after which executing the Multilayer Perceptron involves some matrix multiplications, while Optimal and ModLP have to execute complex Integer and Linear Programming algorithms. The fastest scheduler is DI with 0.22ms spent per workload. DI is very fast because it works with a simple sort mechanism.

6.10.3 Significance of Results

The MLP2 scheduler is better than DI, the state-of-the-art scheduler, in 7 out of 9 test workloads, mostly by a wide margin, and approximately equal to the Optimal Scheduler in 6 out of 9 test workloads, exactly equal in 4. The MLP scheduler produces schedules very close to the theoretical optimum and much better schedules than DI. Also, it has much faster execution time than the optimal scheduler, and relatively close scheduler execution time to DI. The SVM scheduler is better than DI, the state-of-the-art scheduler, in 8 out of 9 test workloads, mostly by a wide margin, and approximately equal to the Optimal Scheduler in 7 out of 9 test workloads, exactly equal in 4. The SVM scheduler produces schedules very close to the theoretical optimum and much better schedules than DI. Also, it has faster execution time than the optimal scheduler, and relatively close scheduler execution time to DI. The reason for such promising results from these advanced machine learning models, as compared to DI, is that DI uses a

simple heuristic, MPI, to schedule tasks, while MLP2 and SVM learn from an optimal scheduling method. Such results are possible with all types of workload, because the optimal scheduling method MLP2 and SVM learn from has knowledge of all possible combinations of tasks on a multicore CPU. Given these performance results, SVM and MLP2 could be used easily as a batch scheduler [4, 2, 3]. In this scenario, tasks (jobs) are submitted for execution in a non-interactive manner; they run to completion once started, without prompting the user for input. Such systems are found in many High Performance Computing (HPC) environments, where large servers exist with many processing units [106, 107]. Batch schedulers for cluster environments include Sun GridEngine [108], Condor [109], and Load Sharing Facility [110]. Current research is focusing on using variants of DI for this purpose, including cluster environments [107]. These studies have shown that DI is superior to the Linux operating system scheduler, even for cluster environments. Our scheduler outperforms DI, with not much additional cost (in terms of run time), so is well suited for this role, be it on single-machine environments or on cluster nodes. MLP2 and SVM could also be modified for integration into an online scheduler, and at various scheduling points the MLP model could be used to reschedule tasks. For example, when a task finishes execution, and when a new task is started. The important issue is that the MLP and SVM models should be trained using training sets representative of the actual workloads which will run on the target machines.

6.11 Conclusion

Our results show that the MLP scheduler is better than DI in 7 out of 9 test workloads, and approximately equal to the Optimal Scheduler in 6 out of 9 test workloads. We show that an MLP scheduler can equal the Optimal scheduler in the average degradation metric and significantly outperform the DI scheduling approach (by more than 10% in most cases). Our results show that the SVM scheduler is better than DI in 8 out of 9 test workloads, and approximately equal to the Optimal Scheduler in 7 out of 9 test workloads. We show that an SVM scheduler can equal the Optimal scheduler in the average degradation metric and significantly outperform the DI scheduling approach (by more than 10% in most cases). Also, an LP scheduler can yield better fairness, thus providing a promising avenue for further research. Both ModLP and MLP are faster

than the Optimal scheduler by a wide margin, but out of the two, only MLP can match the Optimal's scheduling results. SVM is also faster than the Optimal scheduler. MLP and SVM are slower than DI, but makes up for that with far superior scheduling results. Generally, given any set of computer programs, they all behave in the same way. They all have their instructions executed on a CPU, and this requires the use of resources such as the cache and main memory. On current multicore CPUs designs there is resource contention, therefore contention aware scheduling is required. If we were to use a different set of benchmarks which are also non-interactive, these principles are the same. Since our MLP and SVM-based schedulers learn from an Optimal method, which is not dependent on what specific benchmarks are running, but rather on knowledge of performance of all possible processor-allocation combinations, they will be able to achieve comparable results. The greater the difference in co-scheduling performance for various combinations of tasks in a set of benchmarks, the greater the advantage our scheduler designs will have over non-contention-aware schedulers. Our focus was on batch workloads, with non-interactive workloads, however Section 6.10.3 also describes how our schedulers could be used for online scheduling, which could involve running interactive workloads.

Chapter 7

Conclusions

7.1 Achievements

The goal of this thesis was to study the application of optimization and machine learning techniques in improving the performance of operating system schedulers. The operating system schedulers in question included the ones in the Linux 2.6.23 and 2.6.28 kernels, and generic ones implemented in the AKULA [12, 19, 22] scheduling simulator.

The overall work of this thesis involves application of Experimental Statistics, Optimization, and Machine Learning, to a very important software problem: Operating System CPU scheduling. The following contributions have been made:

- A methodology based on Statistical Experiment design has been developed to tune an operating system scheduler for a specific workload. We used Response Surface Methodology (RSM) to solve the problem of tuning the scheduler by using optimal values for the various scheduler parameters. The parameters have wide ranges so using trial and error would require very many cycles of setting parameters, running benchmarks, and evaluating results. Our methodology minimizes the number of required experiments. This means far less benchmark runs. Results show that our methodology achieves much better benchmark run times than without the scheduler tuning.
- A methodology based on Particle Swarm Optimization has been formulated to tune an operating system scheduler. To the best of our knowledge, this approach has never been studied before. Results show much better benchmark run times than without scheduler tuning.

- A methodology based on one-dimensional optimization techniques has been formulated to tune an operating system scheduler. Our methodology makes novel use of relationships within the scheduler to achieve very good results. To the best of our knowledge, this approach has never been studied before. Our methodology required careful examination of the Linux scheduler to discover the relationships. Golden Section method was used. Results show much better benchmark run times than without scheduler tuning.
- The previous two methodologies above have been compared, and their advantages and disadvantages highlighted.
- A novel method for simulating benchmarks in a scheduling simulator was developed: Finite State Machine simulation. This was compared with another state-of-the-art method called bootstrapping [12]. The approach of benchmark simulation was used to evaluate workloads in the global and one-dimensional optimization approaches. To the best of our knowledge, Finite State Machine simulation has not been used for simulating scheduling benchmarks before, and no such comparison has been done before.
- A scheduler focused on multicore scheduling problems was designed, which is faster than the optimal scheduler, while generating schedules which are relatively close to the optimal, outperforming the state-of-the-art contention-aware scheduler [11]. Our scheduler relies on Model 5 Prime, MLP, and SVM machine learning models. Various such models were evaluated in order to achieve the results. This approach is novel and has not been used previously for operating system scheduling.

For our scheduler tuning methods, we focus on throughput, minimizing turnaround time (Equation 4.4). Therefore the sort of programs that work well with these methods are throughput oriented, as opposed to response time oriented where user-interaction is very important, for example. Black Box Learned Optimization can be used for non-interactive workloads (batch) and for online scheduling, which could involve interactive workloads, where response time is important.

7.2 Future Work

A follow up research area to our Operating System scheduler tuning approaches could include further studies on self-tuning schedulers, such that values for tuning parameters are not identified off-line but online, within the scheduler, during execution of workloads. This approach was not pursued because it requires modification of the operating system scheduler, and is therefore specific to a scheduler, and even to a version of the scheduler in question. Also, improvements might be marginal.

A follow up research area to Black Box Learned Optimization is White Box Learned Optimization. Learned Optimization is a process of carrying out optimization using Machine Learning (ML) models and algorithms. Optimization problems are formulated and solved using some optimization algorithm, for example Linear or Integer Programming. Machine Learning algorithms then learn how the optimization is done. When a new problem is tackled, the optimization algorithm is used partially, for example optimization could proceed up to the halfway point, and then the problem's attributes are simply fed into the ML model and a result is produced, without the lengthy process of a full optimization process. The features of the model being mapped to results include attributes of the problem and attributes of the optimization iterations generated on the way to an optimal solution. For example, if the Linear Programming algorithm with Simplex algorithm is the optimization algorithm being observed, on the way to the optimal solution a tableau is generated and manipulated. Therefore variables move in and out of a basis. A snapshot of the state of the basis can be fed into the ML model, together with other problem attributes, and a result is produced.

7.3 Summary

The work described in this thesis cuts across several technical areas, including scheduling, operating systems, computer architecture, machine learning, and optimization. We have tied all these areas together to solve problems prevalent in modern operating systems: process scheduler tuning and processor assignment for multicore systems. We have made significant contributions towards the resolution of these problems.

References

- [1] DTREG. (2012, Aug.) SVM - Support Vector Machines. <http://www.dtrek.com/svm.htm>, [Online. Last Accessed: July 2012]. DTREG. xii, 84
- [2] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. Hoboken, New Jersey: John Wiley & Sons, Inc., 2009. 1, 2, 8, 22, 35, 41, 51, 57, 59, 71, 93
- [3] D. M. Dhamdhere, *Operating Systems: A Concept-Based Approach*. McGraw-Hill, 2007. 1, 2, 8, 22, 35, 41, 51, 57, 59, 71, 93
- [4] A. S. Tanenbaum, *Modern Operating Systems*. Upper Saddle River: Pearson Education, Inc., 2009. 1, 2, 8, 22, 35, 41, 51, 57, 59, 71, 93
- [5] D. M. Zydek, "Processor allocator for chip multiprocessors," Ph.D. dissertation, University of Nevada, May 2010. 2
- [6] S. Johnson, W. Hartner, and W. Brantley, "Improving linux kernel performance and scalability," <http://www.ibm.com/developerworks/linux/library/l-kperf/>, [Online. Last Accessed: March 2010], IBM DeveloperWorks, Tech. Rep., 2003. 2, 9, 35
- [7] J. S. Barrera, "Self tuning systems software," in *Proceedings of the Fourth Workshop on Workstation Operating Systems*. Napa: IEEE, 1993, pp. 194–197. 2, 9, 35
- [8] J. Moilanen and P. Williams, "Using genetic algorithms to autonomically tune the kernel," in *Proceedings of the Ottawa Linux Symposium*. Ottawa: Linux Symposium, Inc., 2005, pp. 327–338. 2, 9, 35
- [9] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff." *Solid-State Circuits Newsletter, IEEE*, vol. 11, no. 5, pp. 33–35, Sep. 2006. 3, 12
- [10] K. Olukotun and L. Hammond, "The future of microprocessors," *Queue*, vol. 3, no. 7, pp. 26–29, Sep. 2005. 3, 12, 13
- [11] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Transactions on Computer Systems*, vol. 28, no. 4, pp. 1–45, Dec. 2010. 3, 4, 7, 13, 15, 31, 32, 72, 73, 74, 76, 79, 87, 96

REFERENCES

- [12] S. Zhuravlev, S. Blagodurov, and A. Fedorova, “AKULA: A toolset for experimenting and developing thread placement algorithms on multicore systems,” in *Proc. of the Nineteenth International Conference on Parallel Architectures and Compilation Techniques (PACT’10)*, Vienna, Austria, Sep. 11–15, 2010, pp. 249–259. 3, 5, 7, 13, 23, 24, 27, 28, 31, 60, 72, 77, 87, 95, 96
- [13] E. Frachtenberg and U. Schwiegelshohn, “New challenges of parallel job scheduling,” in *Job Scheduling Strategies for Parallel Processing*, E. Frachtenberg and U. Schwiegelshohn, Eds. Berlin: Springer Berlin/Heidelberg, 2008, vol. 4942. 3, 72
- [14] S. Siddha, V. Pallipadi, and A. Mallick, “Process scheduling challenges in the era of multi-core processors,” *Intel Technology Journal*, vol. 11, no. 4, pp. 361–369, Nov. 2007. 3
- [15] S. Blagodurov, S. Zhuravlev, S. Lansiquot, and A. Fedorova, “Addressing cache contention in multicore processors via scheduling,” <ftp://fas.sfu.ca/pub/cs/TR/2009/CMPT2009-16.pdf>, [Online. Last Accessed: January 2011], Simon Fraser University, School of Computing Science, Burnaby, BC, Canada, Tech. Rep. TR 2009-16, 2009. 4, 15, 31, 76, 87
- [16] K. Tian, Y. Jiang, X. Shen, and W. Mao, “Optimal co-scheduling to minimize makespan on chip multiprocessors,” in *Proceedings of the 16th Workshop on Job Scheduling Strategies for Parallel Processing*, ser. JSSPP’12. New York, NY, USA: Springer-Verlag, 2012, to appear. 4, 72
- [17] Y. Jiang, X. Shen, J. Chen, and R. Tripathi, “Analysis and approximation of optimal co-scheduling on chip multiprocessors,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT ’08. New York, NY, USA: ACM, 2008, pp. 220–229. 4, 72
- [18] K. Tian, Y. Jiang, and X. Shen, “A study on optimally co-scheduling jobs of different lengths on chip multiprocessors,” in *Proceedings of the 6th ACM conference on Computing frontiers*, ser. CF ’09. New York, NY, USA: ACM, 2009, pp. 41–50. 4, 72
- [19] S. Zhuravlev, “Designing scheduling algorithms for mitigating shared resource contention in chip multicore processors,” Master of Science Thesis, Simon Fraser University, Spring 2011. 4, 5, 31, 72, 77, 95
- [20] Y. Jiang, K. Tian, X. Shen, J. Zhang, J. Chen, and R. Tripathi, “The complexity of optimal job co-scheduling on chip multiprocessors and heuristics-based solutions,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 7, pp. 1192–1205, Jul. 2011. 4, 16, 70, 72, 73, 76
- [21] A. Wierman, “Scheduling for today’s computer systems: bridging theory and practice,” Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, USA, 2007. 5

REFERENCES

- [22] A. Fedorova, S. Blagodurov, and S. Zhuravlev, “Managing contention for shared resources on multicore processors,” *Communications of the ACM*, vol. 53, no. 2, pp. 49–57, Feb. 2010. 5, 95
- [23] Y. Zhang and B. Bhargava, “Self-learning disk scheduling,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 1, pp. 50–65, 2009. 9, 35
- [24] P. J. Teller and S. R. Seelam, “Insights into providing dynamic adaptation of operating system policies,” *Operating Systems Review*, vol. 40, no. 2, pp. 83–89, 2006. 9, 35
- [25] R. Jain, *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc., 1991. 9, 25, 41, 48, 52, 91
- [26] M. W. Totaro, “Multi-hop wireless mesh networks: Performance evaluation and empirical models,” Ph.D. dissertation, University of Louisiana at Lafayette, 2007. 10, 35, 39, 40
- [27] K. K. Vadde, “Network protocols: Interactions and their statistical optimization,” Ph.D. dissertation, Arizona State University, 2005. 10, 35, 40
- [28] P. Shivam, “Cutting corners: Workbench automation for server benchmarking,” in *Proceedings of USENIX’08: The 2008 USENIX Annual Technical Conference*, 2008. 11, 35, 40
- [29] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer, “Using probabilistic reasoning to automate software tuning,” in *Proceedings of the SIGMETRICS/Performance’04 Conference*, 2004, pp. 404–405. 11, 40
- [30] D. G. Sullivan, “Using probabilistic reasoning to automate software tuning,” Ph.D. dissertation, Harvard University, 2003. 11, 40
- [31] K. Olukotun, L. Hammond, and J. Laudon, *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*, ser. Synthesis Lectures on Computer Architecture, M. D. Hill, Ed. Morgan & Claypool, 2007. 12
- [32] Intel Corporation. (2012, Jul.) Intel® Xeon® processor E7-8800 Product Family. <http://ark.intel.com/products/series/53672>, [Online. Last Accessed: August 2012]. Intel Corporation. 13
- [33] T. E. Carlson, W. Heirman, and L. Eeckhout, “Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation,” in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, Nov. 2011, pp. 1–12. 13
- [34] Advanced Micro Devices, Inc. (2012, Jul.) AMD Opteron™ 6200 Series Processors. <http://www.amd.com/us/products/server/processors/6000-series-platform/6200/Pages/6200-series-processors.aspx>, [Online. Last Accessed: August 2012]. Advanced Micro Devices, Inc. 13

REFERENCES

- [35] Intel Corporation. (2012, Jul.) Many Integrated Core (MIC) Architecture. <http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html>, [Online. Last Accessed: August 2012]. Intel Corporation. 13
- [36] R. Illikkal, V. Chadha, A. Herdrich, R. Iyer, and D. Newell, "Pirate: Qos and performance management in cmp architectures," *SIGMETRICS Performance Evaluation Review*, vol. 37, pp. 3–10, Mar. 2010. 13, 75
- [37] E. Hoh, "Vector-based scheduling for the core2-architecture," Study Thesis, University of Karlsruhe, System Architecture Group, University of Karlsruhe, Germany, Jan. 20 2009, <http://i30www.ira.uka.de/teaching/theses/pasttheses/> [Online. Last accessed: June 2011]. 14, 75
- [38] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis, "autopin automated optimization of thread-to-core pinning on multicore systems," in *Transactions on High-Performance Embedded Architectures and Compilers III*, ser. Lecture Notes in Computer Science, Stenström, Per, Ed. Springer Berlin / Heidelberg, 2011, vol. 6590, pp. 219–235. 14, 75
- [39] H. S. Kim, "Bat intelligent hunting optimization with application to multi-objective multiprocessor scheduling," Ph.D. dissertation, Case Western Reserve University, Aug. 2010. 16
- [40] B. Malakooti, S. Sheikh, C. Al-Najjar, and H. Kim, "Multi-objective energy aware multiprocessor scheduling using bat intelligence," *Journal of Intelligent Manufacturing*, vol. 60, pp. 1–15, 2012, 10.1007/s10845-012-0629-6. 16
- [41] J. K. Rai, A. Negi, R. Wankar, and K. D. Nayak, "Characterizing l2 cache behavior of programs on multi-core processors: Regression models and their transferability," in *World Congress on Nature and Biologically Inspired Computing, NaBIC 2009*. Coimbatore: IEEE, Dec. 9–11, 2009, pp. 1673–1676. 16, 74
- [42] —, "On prediction accuracy of machine learning algorithms for characterizing shared l2 cache behavior of programs on multicore processors," in *2009 First International Conference on Computational Intelligence, Communication Systems and Networks*. Indore, India: IEEE Computer Society, Jul. 23–25, 2009, pp. 213–219. 16, 74
- [43] —, "A Machine Learning Based Meta-Scheduler for Multi-Core Processors," *International Journal of Adaptive, Resilient and Autonomic Systems*, vol. 1, no. 4, pp. 46–59, Dec. 2010. 17, 74
- [44] A. W. Apon, T. D. Wagner, and L. W. Dowdy, "A learning approach to processor allocation in parallel systems," in *Proceedings of the eighth international conference on Information and knowledge management*, ser. CIKM '99. New York, NY, USA: ACM, 1999, pp. 531–537. 17
- [45] R. H. Myers, D. C. Montgomery, and C. M. Anderson-Cook, *Response Surface Methodology*. Hoboken, New Jersey: John Wiley & Sons, Inc., 2009. 20, 38, 44, 62

REFERENCES

- [46] Y. Zhang. (2008) Hackbench (CFS scheduler tools). <http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c>, [Online. Last Accessed. April 2010]. 20, 25, 50, 60
- [47] J. Andrews. (2009) Additional CFS benchmarks. http://kerneltrap.org/Linux/Additional_CFS_Benchmarks, [Online. Last Accessed: March 2010]. 20, 25, 50
- [48] ——. (2007) Benchmarking CFS. http://kerneltrap.org/Linux/Benchmarking_CFS, [Online. Last Accessed: March 2010]. 20, 25, 50
- [49] M. E. H. Pedersen, “Tuning & simplifying heuristical optimization,” Ph.D. dissertation, University of Southampton, Southampton, Jan. 2010. 20, 62
- [50] M. Sipser, *Introduction to the Theory of Computation*. Boston, Massachusetts: Thomson Course Technology, 2006. 22, 26
- [51] J. Calandrino, D. Baumberger, T. Li, J. Young, and S. Hahn, “Linsched: The linux scheduler simulator,” in *Proc. of the 21st International Conference on Parallel and Distributed Computing and Communication Systems (PDCCS-2008)*, New Orleans, LA, USA, Sep. 2008, pp. 171–176. 22, 24, 59
- [52] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A full system simulation platform,” *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002. 23
- [53] J. Moses, K. Aisopos, A. Jaleel, R. Iyer, R. Illikkal, D. Newell, and S. Makineni, “Cmp-sched\$im: Evaluating os/cmp interaction on shared cache management,” in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, Apr. 2009, pp. 113–122. 23
- [54] M. M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal, “Analyzing parallel programs with pin,” *Computer*, vol. 43, no. 3, pp. 34–41, Mar. 2010. 23
- [55] A. Kumar, “Multiprocessing with the completely fair scheduler,” <http://www.ibm.com/developerworks/linux/library/l-cfs/>, [Online. Last Accessed: March 2011], IBM DeveloperWorks, Tech. Rep., 2008. 23, 59, 63
- [56] W. Mauerer, *Professional Linux Kernel Architecture*. Wrox Press, 2008. 23, 42, 59, 63
- [57] I. Molnar. (2008) CFS scheduler tools. <http://people.redhat.com/mingo/cfs-scheduler/tools/>, [Online. Last Accessed: March 2010]. 25, 42, 50
- [58] I. Sommerville, *Software Engineering*, 6th ed. Harlow, Essex, UK: Pearson Education Limited, 2001. 30
- [59] J. L. Henning, “SPEC CPU2006 Benchmark Descriptions,” *SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, Sep. 2006, <http://www.spec.org/cpu2006/> [Online. Last accessed: January 2011]. 5, 32, 87

REFERENCES

- [60] S. Zhuravlev. (2011, Jul.) AKULA Multicore Scheduler Simulator. <http://synar.cs.sfu.ca/akula>, [Online. Last Accessed: June 2012]. Simon Fraser University. 32
- [61] M. J. Anderson and P. J. Whitcomb, *DOE Simplified*. Boca Raton: CRC Press, 2007. 36, 37, 38, 44
- [62] —, *RSM Simplified*. Boca Raton: CRC Press, 2005. 38, 43, 44, 52, 56, 62
- [63] M. T. Jones, “Inside the linux 2.6 completely fair scheduler,” <http://http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/>, [Online. Last Accessed: October 2010], IBM DeveloperWorks, Tech. Rep., 2009. 41
- [64] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, Massachusetts: The MIT Press, 2001. 42
- [65] K. Mehlhorn and P. Sanders, *Algorithms and Data Structures: The Basic Toolbox*. Berlin: Springer-Verlag Berlin Heidelberg, 2008. 42
- [66] D. G. Feitelson. (2009) Workload modeling for computer systems performance evaluation. <http://www.cs.huji.ac.il/~feit/wlmod/>, [Online. Last Accessed: March 2010]. The Hebrew University of Jerusalem. 50
- [67] R. Hussey. (2009) Scheduler benchmarks - a follow up. <http://kerneltrap.org/mailarchive/linux-kernel/2007/9/17/261647>, [Online. Last Accessed: March 2011]. 50
- [68] H. K. Toh, J. Wen, and H. Zhiqian, “Linux vm - comparing virtual memory performance between linux version 2.4 and 2.6 on low memory system,” Florida State University, Tech. Rep., 2004. 50
- [69] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee, “Task-aware virtual machine scheduling for i/o performance,” in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2009, pp. 101–110. 50
- [70] M. Dai and Y. Ishikawa, “Delayed processing technique in critical sections for real-time linux,” in *Proceedings of the the 15th Pacific Rim International Symposium on Dependable Computing*, 2009, pp. 269–275. 50
- [71] E. Seo, J. Jeong, S. Park, J. Kim, and J. Lee, “Catching two rabbits: Adaptive real-time support for embedded linux,” *Software Practice and Experience*, vol. 39, no. 5, pp. 531–550, 2009. 50
- [72] J. Jeong, E. Seo, D. Kim, J.-S. Kim, J. Lee, Y.-J. Jung, D. Kim, and K. Kim, “Transparent and selective real-time interrupt services for performance improvement,” in *Software Technologies for Embedded and Ubiquitous Systems*. Springer Berlin/Heidelberg, 2007, vol. 4761/2007, pp. 283–292. 50
- [73] E. di Saverio, M. Cesati, C. di Bagio, G. Pennella, and C. Engelmann, “Distributed real-time computing with harness,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Berlin/Heidelberg, 2007, vol. 4757/2007, pp. 281–288. 50

REFERENCES

- [74] B. Caprita, W. C. Chan, J. Nieh, C. Stein, and H. Zheng, "Group ratio round-robin: $O(1)$ proportional share scheduling for uniprocessor and multiprocessor systems," <http://www1.cs.columbia.edu/~library/TR-repository/reports/reports-2004/cucs-028-04.pdf>, [Online. Last Accessed: August 2010], Columbia University, Tech. Rep., 2004. 50
- [75] K. A. Robbins and S. Robbins, *UNIX Systems Programming*. Upper Saddle River, New Jersey: Prentice Hall PTR, 2003. 2, 51
- [76] M. Sopitkamol and D. A. Menasce, "A method for evaluating the impact of software configuration parameters on e-commerce sites," in *Proceedings of the 5th International Workshop on Software and Performance*, 2005, pp. 53–64. 51
- [77] Stat-Ease Inc. (2009) Design expert. <http://www.statease.com/software.html>, [Online. Last Accessed: March 2011]. 52
- [78] G. E. P. Box, J. S. Hunter, and W. G. Hunter, *Statistics for Experimenters*. Hoboken, New Jersey: John Wiley & Sons, Inc., 2005. 52, 56
- [79] T. Weise. (2009) Global optimization algorithms - theory and application. <http://www.it-weise.de/projects/>, [Online. Last Accessed: September 2011]. 56, 58
- [80] F. J. Solis and R. J.-B. Wets, "Minimization by random search techniques," *Mathematics of Operations Research*, vol. 6, no. 1, pp. 19–30, 1981. 58
- [81] P. Venkataraman, *Applied Optimization with MATLAB Programming*. Hoboken, New Jersey: John Wiley & Sons, Inc., 2009. 56, 58, 65
- [82] E. K. Chong and S. H. Żak, *An Introduction to Optimization*. Hoboken, New Jersey: John Wiley & Sons, Inc., 2008. 56, 58, 62, 63, 65
- [83] R. Hassan, B. Cohanin, O. De Weck, and G. Venter, "A Comparison Of Particle Swarm Optimization And The Genetic Algorithm," in *46th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 2005, pp. 1–13. 58
- [84] E. Elbeltagi, T. Hegazy, and D. Grierson, "Comparison among five evolutionary-based optimization algorithms," *Advanced Engineering Informatics*, vol. 19, no. 1, pp. 43 – 53, 2005, <http://www.sciencedirect.com/science/article/pii/S1474034605000091> [Online. Last accessed: August 2011]. 58
- [85] Y. Guo, W. Li, A. Mileham, and G. Owen, "Applications of particle swarm optimisation in integrated process planning and scheduling," *Robotics and Computer-Integrated Manufacturing*, vol. 25, no. 2, pp. 280 – 288, 2009, <http://www.sciencedirect.com/science/article/pii/S0736584507001342> [Online. Last accessed: August 2011]. 58
- [86] R. V. Lenth, "Response-surface methods in R, using rsm," *Journal of Statistical Software*, vol. 32, no. 7, pp. 1–17, 2009, <http://www.jstatsoft.org/v32/i07/supp/1> [Online. Last accessed: January 2012]. 62

REFERENCES

-
- [87] R Project. (2012) The R project for statistical computing. <http://www.r-project.org/>, [Online. Last Accessed: January 2012]. 62
 - [88] S. Marsland, *Machine Learning: An Algorithmic Perspective*. Boca Raton, Florida: CRC Press, 2009. 71, 73, 80, 83
 - [89] C. M. Bishop, *Pattern Recognition and Machine Learning*, 1st ed. New York, USA: Springer Science+Business Media, LLC, 2006. 71, 73, 80, 81, 82
 - [90] T. O. Ayodele, “Machine learning overview,” in *New Advances in Machine Learning*, Y. Zhang, Ed. 41 Madison Avenue, 31st Fl., Manhattan 10010 New York. USA: InTech, 2010, pp. 19–48, <http://www.intechopen.com/books/new-advances-in-machine-learning/machine-learning-overview> [Online. Last accessed: July 2011]. 71, 80, 82
 - [91] K.-F. Faxén, C. Bengtsson, M. Brorsson, G. Håkan, E. Hagersten, B. Jonsson, C. Kessler, B. Lisper, P. Stenström, and B. Svensson. (2008) Multicore computing - the state of the art. <http://soda.swedish-ict.se/3546/1/SMI-MulticoreReport-2008.pdf> [Online. Last Accessed: July 2010]. Swedish Institute of Computer Science. 71
 - [92] T. Nowotny, M. K. Muezzinoglu, and R. Huerta, “Bio-Mimetic Classification on Modern Parallel Hardware: Realizations on NVIDIA® CUDA™ and OpenMP™,” *International Journal of Innovative Computing, Information and Control*, vol. 7, no. 7(A), pp. 3825–3837, Jul. 2011. 71
 - [93] Y. Sato, T. Miki, and K. Honda, “A Multi-Core Processor Based Real-Time Multi-Modal Emotion Extraction System Employing Fuzzy Inference,” *International Journal of Innovative Computing, Information and Control*, vol. 7, no. 8, pp. 4603–4620, Aug. 2011. 71
 - [94] G. T. Shobha and S. C. Sharma, “Knowledge Discovery for Large Data Sets Using Artificial Neural Network,” *International Journal of Innovative Computing, Information and Control*, vol. 1, no. 4, pp. 635–642, Dec. 2005. 73
 - [95] M. Tabassian, R. Ghaderi, and R. Ebrahimpour, “Handling Classification Problems with Imperfect Labels Using an Evidence-Based Neural Network Ensemble,” *International Journal of Innovative Computing, Information and Control*, vol. 7, no. 12, pp. 7051–7066, Dec. 2011. 73
 - [96] M. R. Othman, Z. Zhang, T. Imamura, and T. Miyake, “A Novel Method for Driver Inattention Detection Using Driver Operation Signals,” *International Journal of Innovative Computing, Information and Control*, vol. 8, no. 4, pp. 2625–2636, Apr. 2012. 73
 - [97] E. Çela, “Assignment problems,” in *Handbook of Applied Optimization*, P. M. Pardalos and M. G. C. Resende, Eds. 198 Madison Avenue, New York, New York, 10016, USA: Oxford University Press, Inc., 2002, pp. 661–678. 78
 - [98] D. W. Pentico, “Assignment problems: A golden anniversary survey,” *European Journal of Operational Research*, vol. 176, no. 2, pp. 774–793, 2007. 78

REFERENCES

- [99] J. Patra and A. Kot, “Nonlinear dynamic system identification using chebyshev functional link artificial neural networks,” *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 32, no. 4, pp. 505–511, aug 2002. 80
- [100] W. Yu, “Nonlinear system identification using discrete-time recurrent neural networks with stable learning algorithms,” *Information Sciences*, vol. 158, no. 0, pp. 131–147, 2004. 80
- [101] V. Vapnik, *The Nature of Statistical Learning Theory*, 2nd ed. Springer, Nov. 1999. 83
- [102] C.-C. Chang and C.-J. Lin, “LIBSVM: A library for support vector machines,” *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>. [Last accessed June 2012]. 84
- [103] C.-W. Hsu, C.-C. Chang, and C.-J. Lin. (2010, Apr.) A Practical Guide to Support Vector Classification. <http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf>, [Online. Last Accessed: July 2012]. National Taiwan University. Taipei 106, Taiwan. 85
- [104] K. J. Kirschner, “Empirical learning methods for the induction of knowledge from optimization models,” Ph.D. dissertation, Georgia Institute of Technology, Aug. 22 2000. 86
- [105] M. Sokolova and S. Szpakowicz, “Machine learning in natural language processing,” in *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, E. S. Olivas, J. D. M. Guerrero, M. M. Sober, J. R. Magdalena, and A. J. S. López, Eds. Hershey, New York: Information Science Reference, 2010, pp. 302–324. 90
- [106] D. Feitelson, L. Rudolph, and U. Schwiegelshohn, “Parallel job scheduling - a status report,” in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Springer Berlin / Heidelberg, 2005, vol. 3277, pp. 1–16. 93
- [107] S. Blagodurov and A. Fedorova, “In search for contention-descriptive metrics in hpc cluster environment,” in *Proceedings of the second joint WOSP/SIPEW international conference on Performance engineering*, ser. ICPE ’11. New York, NY, USA: ACM, 2011, pp. 457–462. 93
- [108] W. Gentzsch, “Sun grid engine: towards creating a compute power grid,” in *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, 2001, pp. 35–36. 93
- [109] M. Litzkow, M. Livny, and M. Mutka, “Condor-a hunter of idle workstations,” in *Distributed Computing Systems, 1988., 8th International Conference on*, Jun. 1988, pp. 104–111. 93
- [110] M. Xu, “Effective metacomputing using lsf multicluster,” in *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, 2001, pp. 100–105. 93