A FLEXIBLE SIMULATION FRAMEWORK FOR PROCESSOR SCHEDULING ALGORITHMS IN MULTICORE SYSTEMS

by

Viswanathan Manickam

B.E., Anna University, Chennai, India, 2006

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
MATHEMATICAL, COMPUTER AND PHYSICAL SCIENCES

THE UNIVERSITY OF NORTHERN BRITISH COLUMBIA

April 2012

Your file Votre référence
ISBN: 978-0-494-94438-7

Our file Notre référence
ISBN: 978-0-494-94438-7

Abstract

In traditional uniprocessor systems, processor scheduling is the responsibility of the operating system. In high performance computing (HPC) domains that largely involve parallel processors, the responsibility of scheduling is usually left to the applications. So far, parallel computing has been confined to a small group of specialized HPC users. In this context, the hardware, operating system, and the applications have been mostly designed independently with minimal interactions. As the multicore processors are becoming the norm, parallel programming is expected to emerge as the mainstream software development approach. This new trend poses several challenges including performance, power management, system utilization, and predictable response. Such a demand is hard to meet without the cooperation from hardware, operating system, and applications. Particularly, an efficient scheduling of cores to the application threads is fundamentally important in assuring the above mentioned characteristics. We believe, operating system requires to take a larger responsibility in ensuring efficient multicore scheduling of application threads.

To study the performance of a new scheduling algorithm for the future multicore systems with hundreds and thousands of cores, we need a flexible scheduling simulation testbed. Designing such a multicore scheduling simulation testbed and illustrating its functionality by studying some well known scheduling algorithms Linux and Solaris are the main contributions of this thesis. In addition to studying Linux and Solaris scheduling algorithms, we demonstrate the power, flexibility, and use of the proposed scheduling testbed by simulating two popular gang scheduling algorithms - adaptive first-come-first-served (AFCFS) and largest gang first served (LGFS). As a result of this performance study, we designed a new gang scheduling algorithm and we compared its performance with AFCFS. The proposed scheduling simulation testbed is developed using Java and expected to be released for public use.

Dedicated to my parents

Acknowledgements

Throughout my time as graduate student at UNBC, I came across many people who have encouraged me in one way or another. I would like to thank them all. Among them, some deserve special thanks.

First of all, I would like to thank my supervisor Dr. Alex Aravind for his continuous support and encouragement. Alex has been a mentor, a well wisher, and a friend. I thoroughly enjoyed working with him, his style and ideas inspired me in getting into research and I enjoy doing that. Next to my supervisor, the people who contributed to my thesis are Dr. Waqar Haque and Dr. Andrea Gorrell. I thank them for their valuable time, effort, suggestions, and encouragement. In addition, I would like to thank the external examiner Dr. Balbinder Deo and Dr. Ajit Dayanandan, chair of my defence for reading my thesis.

I would like to thank my peers and friends who supported me and involved me in informative discussions. First of all, I want to specially thank Hassan Tahir for all his support starting from the day I arrived to Prince George, till I get settled, and also for his brilliant programming ideas which helped me a lot. I thank my fellow graduate students Baldeep, Adiba Mahjabin Nitu, Azizur Rahman, Manoj Nambiar, Narek Nalbandyan, Nahid Mahmud, Fakhar Ul Islam, and Behnish Mann for their support. I thank our computer administrator Heinrich Butow for his continuous support. Finally, I would like to sincerely thank Dr. Mahi Aravind, for the wonderful family dinners and parties on various events and occasions.

Finally, much of the credit goes to my family, especially, my parents who supported me right to the end. A special thanks to my sister Kavitha and my brother in law Ramasamy for encouraging me to apply UNBC. Once again, I thank them all. Thanks Guys!

Viswanathan Manickam

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The scheduling problem in computing systems has been studied in the last several decades. With the recent arrival of multicore processors, the scheduling problem has gained renewed interest. The contribution of this thesis is related to the scheduling problem in multicore computer systems.

The primary goal of computer systems is to execute applications safely, securely, correctly, and efficiently. The hardware and system software have been designed to work together to achieve this goal. Until recently, the developments in hardware and system software have kept pace with each other to meet this goal. Operating system, which has scheduling as its central component, is the major part of system software. Hence, scheduling plays a pivotal role in effective use of computer systems.

A paradigm shift in hardware technology has happened very recently. Instead of increasing speed, the chip designers are starting to put a number of execution cores within a single processor. Such processors are called multicore processors. To exploit the capabilities of multicore processors effectively, the software community requires to make at least two main changes: (i) a new way of designing software for multi-

core processors, and (ii) system software must be redesigned with new capabilities to manage multicore processors. The proposal of this thesis is related to the efforts in meeting the latter demand. Particularly, this thesis proposes to develop a flexible simulation framework to study the performance of scheduling algorithms for multicore processors. The purpose of this chapter is to explain why multicore scheduling is interesting and worthy of investigation. For that, we first briefly describe the recent trends in hardware, computing, applications, operating system, and scheduler.

1.1 Hardware Trend

Nearly forty five years ago, Intel co-founder Gordon Moore predicted that transistor density on integrated circuits will be doubled about every two years. In terms of speed, this law is equivalent to: processor speed will be doubled about every eighteen months. Ever since Moore's prediction, the hardware technology has been driven to produce processors with highest possible speed by increasing the clock rate.

As limited by the laws of physics, the microprocessor design hit the clock cycle wall, and therefore the chip designers had to come up with a new way of exploiting the benefit of Moore's law. The new way is to use the extra transistors to add multiple execution units (referred to as cores) within a single processor. Each core is capable of executing independently of other cores in the processor. This trend in multicore processors indicates that all future processors will be multicore [1].

As Intel, AMD, Fujitsu, IBM, and Sun Microsystems are already shipping their desktops and workstations with multicore processors [2], multicore systems are rapidly emerging as the mainstream computing platforms. This hardware trend seems to continue, and we will have hundreds and even thousands of cores in a single processor in the near future [1]. The abundant availability of execution cores is expected to

revolutionize the way we will design software for these systems in the future.

Another hardware trend, driven by high performance computing groups, has been introduction of various parallel computers. A parallel computer is a single computer with multiple internal processors or multiple computers interconnected to form a coherent high-performance computing platform [3]. Multicore processors have similarity to some parallel computers in structure and functionality, but they differ in several other aspects.

1.1.1 Multicore Processors vs. Parallel Computers

Parallel processors or parallel computers were around as early as single processor systems. In terms of execution units, parallel processors and multicore processors are similar. Multicore processors and parallel processors have two or more execution units. They differ mainly in their purpose, other resources, and application domains. The main purpose of parallel computers is to increase the performance of applications which have longer run times. Parallel processors are often designed for certain types of applications and usually run them in static partitions. In order to utilize the parallel processors effectively, the applications must be parallelized.

In parallel processors, the scheduling is mostly done at the user level rather than taken care by the operating system. Typically, users of parallel computing design their application programs incorporating the logic of scheduling and synchronization. Often, compilers or libraries like OpenMP [4] and MPI [5] help the application designers in achieving this task.

Multicore systems are emerging as general purpose computers, and they are expected to be used in a wide range of domains - from desktop, workstations, and servers. That is, multicore systems are expected to handle multiple types of applica-

tions including interactive workloads, realtime tasks, and normal workloads. Therefore, unlike parallel computers, the task of scheduling in multicore processors cannot be left to applications. The operating system requires to take a major responsibility of scheduling applications to multicore processors. So, there are fundamental differences between parallel processor systems and multicore processor systems [6], and multicore processor systems require redesign of scheduling algorithms. Designing efficient scheduling algorithm for multicore processor systems, we believe, is a complex task.

1.2 Computing Trend

Until recently, most general purpose computing are desktop based and most high performance computing are based on parallel computing and cluster computing [3]. Parallel computing often divides the tasks into smaller ones and uses parallel computers to execute them simultaneously. Cluster computing has the same objective, but the computing infrastructure is a set of loosely connected computers with a suitable software module to coordinate the computers in executing parallel programs. Another computing paradigm called grid computing [7] has taken this trend one step further by pooling computing resources from multiple administrative domain to solve a single problem. The most recent trend is cloud computing. Cloud is a computing infrastructure paradigm that offers computation and storage as web-based services [8]. Cloud computing typically has parallel computers and/or cluster computers as its server nodes. With the emergence of multicore processors, the future clouds are expected to have multicore blades as their servers [8, 9].

1.3 Applications Trend

In the 1980s, only a small group of people knew about computers, and even a smaller group of people used one. Now, almost everyone knows what a computer is,

most of us use it on a daily basis for reading news, listening to music, etc. Thus, the use of computing continues to infiltrate every aspect of our life and automation in the practical world continues to increase. With the recent computing trends of cloud computing and mobile accesses, and the access to social media and Internet, the need for more automated services are expected to be accelerated. That means more software for applications are expected to be developed, and they have to be developed in a way to effectively run on the future computing platforms.

1.4 Operating Systems Trend

Operating systems is one of the core areas in computer science, and has accumulated a large body of literature. However, most of the work in the past was done either in the domain of uniprocessor systems, where the scheduler has the full responsibility of managing applications including scheduling, or in the domain of parallel processing, where the responsibility of task scheduling is largely left to application designers[1].

Operating system is one of the most complex software systems, and designing one is a challenging task. It has an influence on almost all other systems, both hardware and software that involve computing. Operating systems spend a huge portion of their time in executing applications [10].

Generally, the functionalities of operating systems add and evolve constantly to meet the needs of new technologies and applications. However, the operating system design related to processor management has not been changed much in the last several years as the number of processors has not been changed much. Current operating systems were designed for single or few cores. Thus, most developments have been

[1]Parallel processing typically involves complex problems requiring high computational time. Based on the software specifically designed for parallel programming, the program designers divide a complex problem into component parts and then assign the component parts to be executed on individual processors [3].

in the domain of file system, security, device management, and user interface. These developments are not aimed to meet the demand of multicore processors.

The traditional approach of building an operating system for every individual hardware model is no more acceptable, because it will be out of date once new hardware arrives [11]. Some of the main issues of current operating system are:

1. **Scalability:** Current operating systems are not designed to be scalable for multicore processors. Therefore, adding hardware resource would requires redesigning of operating system, as the current operating system cannot utilize the newly added hardware to increase its efficiency [1].

2. **Resource allocation:** Current operating systems are designed for computation with limited resources. This may not be the case of multicore systems. Since multicores require abundant resources for their computation, such resources are expected to be added accordingly for effective execution. For example, with the recent advancements of hardware multilevel caches, inefficient allocation of cache will result in performance degradation [12].

3. **Parallelism:** If the current trend of multicore processor continues, the workload of an operating system managing the number of cores will continue to increase. Dividing the core management workload and handling concurrently require parallelism in kernel level. Parallelizing the kernel is difficult and marginally successful. This pushes us to seek new approaches [13]. Simply tuning applications to get advantage of the available cores may not be good enough when there is a mass deployment of cores. Therefore, the operating systems for the future multicore systems have to be redesigned or developed to effectively manage the cores among the applications to achieve/exceed the expectations of the users [14].

As multicore systems offer more cores, handling these cores to serve the applications is becoming more challenging.

1.5 Schedulers Trend

To meet the requirement of multicore systems, the most important component of operating system that might need a radical change is the processor scheduler. Most developments in the operating system domain in the last several years have been on file system, security, device management, and user interface, and only minimal changes have been proposed for schedulers. These schedulers have been focusing on effectively multiplexing the CPU among the competing processes to assure fairness, quick response, and minimize the turnaround time.

The traditional operating systems such as Linux, Windows, and Solaris schedule the processes using time multiplexing. The approach of time multiplexing alone is not suitable for multicore schedulers for several reasons. First of all, with the advancements of hardware and the availability of multilevel cache hierarchy, scheduling a core to a thread exclusively could reduce the latency and hence increase the performance. That is, the thread scheduled alone in a core can effectively use the cache to reduce its execution time. Secondly, time multiplexing does not effectively deal with distributing the work among different cores so that no core sits idle when there is heavy workload on peer cores. Finally, time multiplexing does not reduce the impact of access to cache and DRAM, which are considered expensive operations. Rather, it could increase the overhead on accessing shared resources such as cache, memory, and network. These reasons force us to re-think the scheduler design.

The operating system literature on multicore systems is relatively limited and most of the publications are within the last five years [1, 6, 8, 10, 12–36]. In that, only

a small portion is related to multicore scheduling algorithms [12]. Most of them are related to effectively dealing with resource contention.

The proposed approaches to design scheduling algorithms for multicore systems vary greatly and differ in their recommendations. One view is that multicore systems have in fact simplified the scheduling. That is, since we have plenty of cores, there is no need to worry about intricate time multiplexing strategies, simply giving enough cores to applications will simplify the scheduling. On the other extreme, several researchers feel that multicore systems have complicated the scheduling task, as it needs to consider several factors such as cores, caches, networks, and application requirements together in offering best possible service. A number of work suggest ideas in between these extreme cases. Most of these ideas are related to cache contention. Again, related to resource contention, there are two views. One group strongly advocates to incorporate contention aspects into the scheduling algorithms [12,23]. Another group argues to decouple contention management from scheduling [34,36]. There is another direction of research that explores the question of whether to keep operating systems and applications together or separately [13,22].

Overall, the field of multicore scheduling is very young and the proposed ideas on multicore scheduling are preliminary.

1.6 Where do the contributions of this thesis fit in?

From the above discussion, we infer that the software systems that worked well for sequential systems might not effectively work with the multicore systems. Therefore, there is a gap between the rapidly emerging hardware technology and relatively slow software technology. The recent research trend indicates that the software systems, particularly the operating system must be redesigned to reduce this gap. More

importantly, new scheduling algorithms must be developed to utilize the multiple resources offered by multicore systems. This thesis is an effort to help achieve this goal. More specifically, this thesis contributes to help develop new scheduling algorithms for multicore systems.

The most difficult aspects of developing a novel scheduling algorithm are implementing and testing its performance [25]. We believe a flexible multicore scheduler simulator framework with proper support for simulation and testing would be very useful. Developing such a comprehensive framework is the primary goal of this thesis.

1.7 Thesis Organization

The fundamentals of multicore scheduling and the related work are presented in Chapter 2. Next, in Chapter 3, we present the motivation, contributions, and research methodology. In Chapter 4, we present the design and the implementation of the multicore scheduler simulation framework. The framework and its implementation are the major contributions of this thesis. We present the implementation and simulation study of Linux and Solaris scheduling algorithms in Chapter 5, and a new gang scheduling algorithm is presented and its performance compared to two well known gang scheduling algorithms in Chapter 6. Finally, in Chapter 7, we conclude the thesis and list some future directions to extend the work carried out in this thesis.

Chapter 2

Multicore Scheduling

Scheduling is a fundamental problem in several systems. In processor scheduling, threads are assigned to processors for execution with the objective of optimizing certain performance metrics such as maximum throughput, minimum average response time, minimum average waiting time, and/or maximum CPU utilization. Threads are schedulable entities which achieve the intended tasks by their execution. Cores are physical execution units. In a multicore context, scheduling can be viewed at two levels: balancing the system load among the cores and multiplexing threads on a single core. In actual implementations, these two tasks could be integrated as one scheduling module.

A scheduling algorithm is a set of rules that define how to select the next thread for execution. This problem is well studied in single processor(core) context and numerous scheduling algorithms exist in the literature [37,38]. Present multiprocessor operating systems such as Linux and Solaris use a two-level scheduling approach [12]. In one level the scheduler balances the load across cores, and in another level the scheduler uses a distributed run queue model with per core queues and local scheduling policies

to manage each core.

2.1 Load Balancing

In general, load balancing in multicore or homogeneous multiprocessor systems can be done in several ways [12, 39]. In one extreme, all the jobs can be kept in one shared global queue and schedule a job from this queue whenever a core becomes free. This approach is simple and balances the workload effectively, and hence appears to increase the core utilization. But, in reality, this approach degrades the performance due to cache pollution as there is a high probability of jobs frequently migrating from one core to another. Modern systems achieve high performance by effectively exploiting the locality of reference, and by keeping frequently accessed data in local cache. Job migrations rarely utilize this benefit, and hence a significant amount of time is spent on accessing data from farthest locations such as last level cache and memory.

On the other extreme, each core can maintain its own queue of jobs to better manage cache affinity and other local resources. This case allows several load balancing strategies by migrating jobs from one local queue to another [39]. There are four simple approaches. The first one is called sender-initiated policy in which lightly loaded cores initiate requests for jobs from other cores. This technique is also referred to as *work stealing* from other cores. In the second approach, called receiver-initiated policy, the heavily loaded cores request other lightly loaded cores to take jobs. The third approach is the combination of both sender-initiated policy and receiver-initiated policy, and therefore called symmetric policy. The fourth one is that the heavily loaded core simply chooses a random destinations to migrate some of its jobs. This simple strategy is found to be working well in practice.

The above general load balancing schemes are applicable only for systems with homogeneous processors, and are not suitable for parallel applications or the types of jobs which are heterogeneous with differing priorities.

2.2 Scheduling Approaches

In single processor system, scheduling of different jobs on a processor is typically done by time sharing. The basic idea of time sharing scheduling is that the processor time is divided into chunks of time called time quanta, and each application executes in different time quanta to complete its task. The critical factor affecting this technique is the time quanta, say, q. When a q is set very large, the applications run longer to complete their executions. When q is set smaller, the applications interleave frequently. The popular time sharing technique with effective interleaving executions is round robin scheduling [37, 38].

Almost all uniprocessor scheduling algorithms used in modern operating systems are time sharing. Among the time sharing algorithms, the most practical algorithms use multilevel feedback scheduling strategy. The basic idea behind multilevel feedback algorithms is that they use different priority queues to manage jobs with varying importance, and the jobs move between queues as their priorities change. The jobs with a lower priority will be served only if the higher priority queues are empty.

The scheduling algorithms used by popular operating systems such as Linux, Solaris, Mac OS, and Windows are some sort of multilevel feedback algorithms. These algorithms with suitable load balancing technique have been adapted for multicore processors. (For this thesis, we have simulated Linux and Solaris schedulers.)

The orthogonal technique to time sharing is space sharing, and it is applicable only in multiprocessor systems. It is an effective generic approach of scheduling multi-

threaded applications on multiprocessor systems. The basic idea is that different applications use different sets of processors during their lifetime. That is, the scheduler dedicates a set of processors to an application for its entire lifetime. Although this technique might offer excellent service to the applications, it may not be good for the utilization of system resources. In this approach, the processor will be idle when the application goes for I/O, or waiting for an event or synchronization.

An effective variant of space sharing approach to parallel applications is that, instead of dedicating a set of processors to an application for its lifetime, parallel threads of an application are scheduled together for a fixed period of time. This technique, originally called co-scheduling later referred to in the literature as gang scheduling, was introduced by Ousterhout [40]. Gang scheduling efficiently uses busy waiting for frequent synchronization. In the literature, frequent synchronization is also referred to as fine-grained synchronization. The idea behind gang scheduling is simple that threads of a same process are scheduled together as a 'gang' on distinct processors so that they can progress in parallel and synchronize with minimal busy waiting involved. A gang is an application containing a set of parallel threads that frequently communicate with each other.

During their executions, threads in a gang communicate for synchronization and data exchange. Often, a thread in a gang cannot proceed further without sufficient progress from other threads. Such threads either do busy waiting or block themselves by suspending from execution until other threads progressed enough. A long busy wait on a processor wastes its execution time. On the other hand, suspending and resuming processes often are also not good, when only a small wait is needed. Blocking results in context switches, which are costly. For several applications inducing small waits, research shows that a busy wait is better than blocking. Gang scheduling algorithms are typically designed to exploit the above observation.

In gang scheduling approach, different applications can use the same set of processors in different time quanta, and same application can use different sets of processors in different time quanta. This is an effective technique that provides an excellent service to parallel applications with threads involving similar loads and fine-grained synchronization.

Using our scheduling simulator developed for this thesis, we study two popular gang scheduling algorithms and propose an improved gang scheduling algorithm. We believe that the proposed algorithm can be used for scheduling gangs in cloud computing, as explained next.

2.2.1 Gang Scheduling in Multicore and Cloud Computing

Recently, it is predicted that the next decade will bring microprocessors containing hundreds, thousands, or even tens of thousands of computing cores, and computational clusters and clouds built out of these multicore processors will offer unprecedented quantities of computational resources [8,22,41]. We discuss the relevance of multicore scheduling in cloud computing assuming that the above prediction will come true.

Cloud computing is a service oriented computing paradigm. It is designed to provide services such as computation, software applications, data access, data management and storage resources to customers through internet transparently [8]. As cloud computing offers computing as a service, customer satisfaction about the services they receive is extremely important. Customer satisfaction is mainly related to cost, fairness, and quality of service. In that, fairness and quality of service are often related to system performance. Particularly, these metrics are primarily influenced by the execution of applications in the cloud. That, in turn, heavily depend on the

processor scheduling in the cloud. That is, to offer services effectively to customers, the service requests must be properly mapped to the available computing resources in the cloud. This is simply a scheduling problem, and it generally involves sequencing and assigning a set of applications on one or more processors (servers) so that the intended criterion is met, while maintaining the maximum possible utilization of system resources. Therefore, processor scheduling is a fundamental problem in cloud computing as it is involved in almost all services that the cloud can offer. As the servers of the cloud are expected to be built from multicore processors, scheduling of multicore processors is an integral component of cloud scheduling [8, 22, 41].

Among the applications of cloud computing, a considerable portion of applications are expected to be from high performance computing groups. Such applications require huge computational resources. Some of these applications are typically designed as parallel threads with frequent synchronization among themselves. These applications are basically gangs. A recent research suggests that gang scheduling can be effective in cloud computing [42].

2.3 Related Work

To set the context for our work, we reviewed the work on operating systems for multicore processors. In this section, we review the work specifically related to our contributions. This thesis has contributions relating to multicore scheduling simulation, performance study of Linux and Solaris scheduling algorithms, performance study of three gang scheduling algorithms, and fairness aspect of gang scheduling algorithms. Next, we review the work related to these contributions.

2.3.1 Scheduler Simulators

A number of simulators for multiprocessors have been proposed in the literature [25, 26, 28, 43–47]. Among them, Simics [28], SimOS [46], SimpleScalar [43], and AMD SimNow [47] emulate the processor at the instruction set level. They differ in emulating different architectures and simulating other components such as I/O and network.

Simics simulates the hardware which can run unmodified operating systems such as Solaris, Linux, Windows XP, and Tru64. Simics supports the following processor models: Ultrasparc, Alpha, x86, x86-64, PowerPC, MIPS, IPF, and ARM. In addition, Simics simulates the device models well enough to execute the device drivers.

SimOS simulates the hardware components to boot, study, and run IRIX operating system and the application that runs on IRIX. SimOS fastens the simulation by changing the mode of execution. There are three modes of execution proposed in SimOS - emulation, rough characterization, and accurate mode. Emulation mode models the hardware that are required to execute the workloads, leaving other uninteresting execution such as booting the operating system, reading from the disk, and initializing the workload. This is the fastest mode. The rough characterization mode approximate the behavior of the system by simulating those uninterested executions. This mode is two or three times slower than the emulation mode. The accurate mode emulates the complete system, and therefore it is the slowest and very time consuming. The accurate mode can be used for measuring the accuracy of the system under simulation.

AMD SimNow simualtes the AMD family processors. SimpleScalar simulates a close derivative of MIPS architecture. Turandot [44] emulates PowerPC. A recent

simulator called COTSon [45] uses AMD SimNow, and employs a statistical sampling technique that can selectively turn on and off the simulation to reduce the overall simulation time.

Since all these simulators emulate machine instructions completely, they all have fine grained accuracy at instruction level. Some of them are used as virtual machines. However, they are very slow as they have to interpret each machine instruction at software level. For example, SimOS - the fastest among the group - can execute workloads only less than 10 times slower than the underlying hardware. Note that SimOS simulates other components to attain this speed. Also, as these simulators will run on host operating system, the scheduling of host operating system will further slow down the execution time of the instruction. Such fine-grained simulators are more suitable for studying low level functionalities of the processors.

These simulators are not suitable for rapid simulations aiming to get quick insight and guidance to develop new scheduling algorithms for future multicore processors. They are machine dependent and emulates only existing hardware. Implementing a scheduling algorithm in a system supported by a regular operating system is hard. Therefore, simulating a new scheduling algorithm for performance study in these simulators is time consuming and hard.

A simulator of Linux scheduler called Linshed was proposed in [26]. This simulator was designed by making changes to original Linux kernel and it runs in user mode. The objective of Linshed was to study the Linux scheduler in depth and was not intended to implement any new scheduling algorithm or comparing with any existing scheduling algorithm.

Recently, a toolset called AKULA [25] was proposed to study scheduling of threads on multicores so as to reduce their cache contention. AKULA toolset assumes the

availability of the task profile termed as *bootstrap data* on cache behavior. Such cache behavior can be obtained only through actual execution on a dedicated multicore system. The bootstrap data contains two information: solo execution time and degradation matrix. Solo execution time is measured when the thread runs alone in the real machine and the degradation matrix is the degradation value from their solo execution time when a thread is scheduled with other threads in different cores which share the cache. For example, there are two threads A and B which are scheduled in two core system. The degradation matrix contains degradation value of of thread A when thread B is scheduled in another core and vice versa. Suppose the degradation value of thread A when scheduled with thread B is 0.75, then the slow down percentage is 75.

Threads on a multicore system can be scheduled in a number of ways. Consider a system with two cores and two level cache memories L1 - local to each core, and L2 shared by both. In this system, all threads can be scheduled to one core or they can be distributed between two cores in several ways. These different ways of scheduling will have different influence on both L1 and L2 level caches. To observe the cache behavior, we need to collect the cache data in all possible ways of scheduling, which will result in large number of combinations.

AKULA collects the cache behavior for a limited set of scheduling combinations. It assumes that the threads scheduled in the cores are allotted with full L1 cache and observes the L2 level cache effect. That is, time sharing on a core is not allowed. For example, there are four threads, say *A, B, C,* and *D*, need to be scheduled and there are two cores in the system which share L2 cache. The degradation matrix will have the degradation value for the following 12 combinations of threads: AB, AC, AD, BA, BC, BD, CA, CB, CD, DA, DB, and DC.

If the number of cores is increased to 3, then a total of 24 combination of data have to be collected. So, the size of the data increases drastically with increasing number of cores as well as increasing number of threads. Also, executing these tasks in real machine to collect the bootstrap trace is tedious and very time consuming task. More importantly, AKULA only runs on the profile data created by actual executions, it cannot be used for testing new workloads. Therefore, it limits its usage only to the system and the workload for which the trace is collected. Even changing a single parameter such as speed, number of cores, cache, workload will make the trace unusable. Due to this restriction, the use of AKULA is very limited.

2.3.2 Fairness and Performance Issues in Gang Scheduling

Gang scheduling has been extensively analysed and several studies have concluded that gang scheduling is one of the best approaches for parallel applications, and hence several gang scheduling algorithms under different conditions have been proposed in the literature [40, 42, 48–55]. Among them First Fit (or First-Come-First-Served (FCFS)) and Best Fit (or Largest-Gang-First-Served (LGFS)) are popular [53].

When enough processors are free, FCFS chooses the job at the head of the queue to schedule and LGFS chooses the largest job in the queue to schedule. FCFS assures high fairness, but does not guarantee the best processor utilization. Consider that a larger gang G is in the head of the queue, and several other smaller gangs are waiting behind G. Assume that there are not enough processors to schedule G, but several other processes from the queue can be scheduled. Now, in FCFS, these processors will be idle until enough processors become free and G is scheduled. Such situations will not only make the processor utilization low, but also have the potential to increase the average waiting time. To avoid such situations, a modification called adaptive FCFS (AFCFS) was introduced [53]. When a gang in the head of the queue cannot

be scheduled, AFCFS schedules other gangs behind in the queue. All these algorithms are susceptible to starvation. To avoid starvation, these algorithms adopt the policy of migrating jobs from processor to processor [53]. Such task migration may not be effective for multicore systems.

Task migration is hardly possible between two heterogeneous multicore systems, and generally expensive even between two homogeneous systems [39, 56]. Also, an efficient task migration can reduce the wait time, but it does not guarantee to eliminate starvation. Therefore, a simple solution to avoid starvation in gang scheduling is an interesting open problem.

Avoiding starvation is an interesting theoretical problem. But, for practical applications, a better fairness measure than free of starvation is most desirable. We found that no such fairness metric has been introduced and used in this context.

Regarding performance, the most widely used metric in the processor scheduling context is average response time. We believe a predictable performance is more valuable than better average response time[1].

2.3.3 Performance Study

We have conducted performance studies on two sets of scheduling algorithms. Next, we discuss the work related those studies.

2.3.3.1 Linux vs. Solaris Scheduling Algorithms

Linux and Solaris schedulers have been constantly tuned and updated [12, 57]. Even the most popular O(1) Linux scheduler introduced in version 2.6 was initially

[1] "it is more important to minimize *variance* in the response time than to minimize the average response time. A system with reasonable and *predictable* response time may be considered more desirable than a system that is faster on the average but highly variable. However, little work has been done on CPU scheduling algorithms to minimize the variance." [37]

expected to be used for a long time, has been overshadowed by the introduction of a completely fair scheduler (CFS) [12, 58]. We decided to study the performance comparison between Linux CFS and the Solaris 10 scheduler.

To the best of our knowledge, we could not find a performance comparison between Linux and Solaris schedulers. Even the complete description of the scheduling algorithms of these operating systems are not comprehensively described in one place. We put a lot of effort to construct the complete algorithm in bits and pieces from various sources for our simulation study.

2.3.3.2 AFCFS vs. LGFS Gang Scheduling Algorithms

Among the popular gang scheduling algorithms, AFCFS alleviates the problem of low processor utilization, and performs better than LGFS under light loads. The performance of these two algorithms have been studied in [51–53]. LGFS, on the other hand performs better than AFCFS under heavy loads.

2.4 Summary

In this chapter, we explained multicore scheduling in a higher level, and looked at load balancing and scheduling approaches. Then, we discussed an interesting class of parallel job scheduling called gang scheduling and its relevance to multicore and cloud computing. With this background, we are ready to present the motivation and contributions of the thesis.

Chapter 3

Motivation and Contributions

This chapter presents the motivation and contributions of this thesis, and the methodology used. This thesis contains three main contributions: (i) a multicore scheduler simulation framework and its implementation; (ii) simulation studies of two popular scheduling algorithms to illustrate the use of the proposed scheduler simulation framework; and (iii) a new scheduling algorithm and its performance study. We start with the motivation for the first contribution.

3.1 Multicore Scheduler Simulation Framework

From the literature study presented in Chapter 2, we identify two potential choices for our thesis work: (a) design and propose a new or improved multicore scheduling algorithm; and (b) design and propose a multicore scheduler simulation testbed where any new scheduling algorithm can be easily simulated and analysed.

The first choice is from the observation that, as the field is young, and there are no widely accepted concrete multicore scheduling algorithms have been proposed in the literature, there is a good possibility of inventing an efficient multicore scheduling

algorithm. The second choice is from the observation that, as the trend in processor technology indicates that the future systems will have hundreds and thousands of cores, any new algorithm proposed for such systems must be studied carefully using a large number of cores using proper set of experiments before it can be adopted for practical use.

After several brainstorming discussions, we started to realize that both choices are risky and challenging as they have open ended goals. However, we felt the second choice has the potential to impact widely, and therefore we chose to proceed in designing and implementing a flexible multicore scheduler simulation framework.

Designing and implementing a multicore scheduler simulation framework involve several research questions to be explored, and some of them are:

- What would be the main purpose of the framework?
- What would be the components of the framework?
- How accurately can the components be modeled?
- What is the level of accuracy that we want in modeling the components of the framework?
- To illustrate the use and flexibility of the framework, which scheduling algorithms can we implement and study?
- What kind of simulation experiments we would like to conduct?

From the literature, we understood that no simulator could replace a real system. However, the design and implementation choices vary greatly depending on the cost and accuracy. Here, the cost is a function of effort, time, complexity, and performance.

The accuracy depends on the purpose of the simulation. Our design choice of the scheduler simulation framework is mainly motivated from the following observations:

- "Simplicity is the key to understanding. ... Simplified simulations provide the best grounds for extracting major properties quickly. ... Simulations done with realistic physical layers normally lead to investigating phenomena with too many variables, too many puzzles, leading to too few explanations, and too few hints for future progress." [59].

- "So, in practice, models that attempt to be highly accurate end up running very small "toy" workloads." [28].

- "... the biggest difficulties in scheduling algorithm development: the difficulty of implementation and the duration of testing. The difficulty of implementation refers to the time and effort needed to convert an idea into the actual code ... The difficulty of implementation and the duration of testing make it infeasible to explore many different scheduling algorithms." [25].

These observations motivated us to design a multicore scheduler simulation framework that is simple but flexible and comprehensive, so that the design space of the scheduling algorithms for multicore systems can be explored rapidly.

Our design objective of the framework is mainly to provide accuracy sufficient to gain initial insights into the performance of the scheduling algorithm under study. We believe such insights will be valuable to guide the researchers in developing new scheduling algorithms with specific objective in mind. As a matter of fact, we encountered such an opportunity of developing an improved scheduling algorithm for a specific class of parallel applications called gangs. Next, we briefly explain how we

were motivated to study gang scheduling algorithms and propose an improved gang scheduling algorithm.

3.2 A New Gang Scheduling Algorithm

Cloud computing is an emerging class of computational platform that has the potential to provide unprecedented compute capacity to future organizations and average users [8]. The trend in multicore processors indicates that all future processors will be multicore, and hence the future cloud systems are expected to have their nodes and clusters based on multicore processors [60]. So the processor scheduling in the future systems will most likely be all multicore processor scheduling. Therefore, multicore scheduling is fundamental to future cloud computing performance. Also, due to multicore revolution, a considerable portion of large applications will be parallel programs. From the literature, we can see that gang scheduling is a dominant strategy to schedule parallel programs with the requirement of frequent synchronization.

Among the popular gang scheduling algorithms, AFCFS alleviates the problem of low processor utilization, but is susceptible to starvation. LGFS, as claimed in the literature [42], outperforms AFCFS in large loads, but again is susceptible to starvation. To avoid starvation, these algorithms adopt a process migration policy. Process migration in this context is migrating gangs between multicore systems. Migrating gangs may not be even possible between two heterogeneous multicore systems, and generally expensive even between two homogeneous systems [39,56]. Gang migration could reduce the overall wait time of the migrating gang, but it does not guarantee to eliminate starvation.

These observations raise a question. Can we design a gang scheduling algorithm with the following characteristics?

1. Freedom from starvation.

2. Predictable and acceptable response.

3. Better processor utilization.

Since LGFS favors large gangs, the smaller gangs are susceptible to starvation or will have longer wait time. This is unacceptable particularly in cloud environment where customer satisfaction hugely depends on fairness and predictable response time. In practice, the customers who receive a little faster service (at the expense of others' long wait) may not be overly satisfied [61]. But, the customers who experience unpredictably long delay, on the other hand, will readily notice the unfairness and unpredictable response and that could potentially drive the cloud business in a negative direction. Therefore, in addition to fast response and high processor utilization, minimal variance in response time is extremely important for quality of service in cloud systems.

3.3 Contributions

This thesis has contributions in the following three categories:

1. **Inventive**

   - A new multicore scheduler simulation framework
   - A new gang scheduling algorithm with increased fairness

2. **Creative**

   - A multicore scheduler simulator (expected to be released as open source software)

3. **Experimental**

- Performance study of Linux and Solaris scheduling algorithms
- Performance study of two popular gang scheduling algorithms AFCFS and LGFS
- Performance study of a new gang scheduling algorithm and comparison with AFCFS

At another angle, this thesis has contributions in theory, algorithm, implementation, and experiments. The scheduling framework is a theoretical abstraction of the scheduling system. The new gang scheduling algorithm is an algorithmic contribution. The scheduling simulator is an implementation of the framework. Finally, the experimental study are the performance study of five scheduling algorithms - Linux and Solaris scheduling algorithms, and three gang scheduling algorithms.

These contributions have several benefits. The insights obtained from the experimental evaluation will help: (i) the users to effectively exploit the hidden power of the above mentioned schedulers, and (ii) the researchers to design new efficient multicore scheduling algorithms, by combining the best ideas of the above studied algorithms, and perhaps adding new ideas. The simulation tool can be used to evaluate any newly designed multicore scheduling algorithm under various conditions before it is adopted for real systems.

The proposed gang scheduling algorithm is simple, fair, and gives predictable performance. Such a predictable performance is attractive from the service point of view. Also, the algorithm is scalable as it solves the starvation problem locally without using process migration. High performance, fairness, and scalability are attractive properties for cloud computing. Therefore, the algorithm is applicable for cloud systems

built from multicore processors.

3.4 Research Methodology

The methodology we followed is generally referred to as constructive research methodology, where the construct could be a new theory, algorithm, model, software, system or a framework. This methodology is most common in engineering and computer science, particularly in systems research. Constructive research methodology involves innovative modeling, design, implementation, and experimentation.

Multicore scheduling is a challenging problem, and we determined to explore the problem in a systematic fashion, using a combination of theory and practice, with an experimental approach. First, the focus is on thoroughly understanding the theory behind processor scheduling by studying and evaluating existing scheduling schemes. Second, based on this understanding of the literature, a multicore scheduler simulation framework with components that can be useful in implementing a new multicore scheduling algorithm is designed and implemented. Third, the components necessary for studying the performance of multicore scheduling algorithm are determined and added to the framework. Finally, the functionality of the proposed framework is demonstrated by simulating five scheduling algorithms, and then illustrating the performance of the simulated scheduling algorithms through performance monitoring and performance metrics. The five major steps involved in the methodology are:

1. Literature survey
2. Design and implementation of the multicore scheduler simulation framework
3. Identification and implementation of performance monitoring components and performance metrics

4. Implementation of three popular classes of scheduling algorithms: Linux, Solaris, and Gang scheduling

5. Conducting simulation experiments on five scheduling algorithms and illustrating their performance results

3.5 Summary

In this chapter, we presented the motivation for our contributions and listed the contributions. It also contains the research methodology that we followed. With this background, we are ready to present the major contribution of thesis - the multicore scheduler simulation framework in the next chapter.

Chapter 4

Multicore Scheduler Simulation Framework

This chapter presents the architecture of the multicore scheduling simulation (MSS) framework, which is the major contribution of this thesis. The primary use of this framework is to facilitate researchers in implementing and simulating multicore scheduling algorithms to evaluate the performance visually and statistically. Before presenting the framework, we briefly explain the simulation technique we used.

4.1 Simulation

Computer simulation is a technique to model and observe the behavior of some real or imagined system over time [62]. A simulator is simply a computer program that transforms the *state of the system* in discrete time points over a specified period of time. Simulation is widely used to study the dynamic behavior of complex systems.

Based on how the system state is modeled and simulated, computer simulations are classified either as continuous or discrete. If the state variables change continuously

over time, then it is called a *continuous* simulation, and if the state variables change only at discrete times, then it is called a *discrete* simulation. In reality most systems are a combination of both. However, depending on the purpose, most systems are simulated either as continuous or discrete, and rarely as hybrid of both.

Discrete simulation is further divided into *time-stepped* and *event-driven*, based on the advancement of simulation time and the update of the system state. In time-stepped simulation, the system state is updated at every time step. In event-driven simulation, the system state is updated at the occurrence of events.

Discrete event simulation consists of an events list, a simulation clock, and an event scheduler. For instance, in simulating the behavior of a queue at the bank-teller, the number of customers arrived and the number of customers served are state variables and they will be updated on the occurrence of the events in the system. The number of customers arrived will be updated when the customer arrives in the bank, and the number of customers served will be updated when the bank-teller serves the customer. Simulation continues until either the events list becomes empty or the simulation time ends.

In discrete event simulation, the simulator maintains a queue of events (also called *event list*) sorted by the simulated time they should occur. *Simulation clock* maintains the simulation time and it is advanced to the time of occurrence of next event in the event list. Since, it is not important to execute the simulation in real-time, the advancement of the simulation time can be the same, faster, or slower than real-time. For example, in the simulation of humans evacuating a building, the queues buildup can be visualized faster than real-time. The current flow through an electric circuit can be simulated slower than real-time, and in-training simulations (for example, flight simulators) can be exhibited real-time speed. An *Event scheduler* executes events

from the events list and the system state changes at the occurrence of each event in the system.

We use both discrete time stepped and discrete event simulation to implement different components of our framework.

4.2 Multicore Scheduler Simulation

From systems point of view, multicore scheduling essentially has two tasks - maintaining the load among the cores and multitasking threads in each core. Generally, the first task is referred to as *load balancing* and the second task is referred to as *local scheduling*. We maintain these abstractions in our scheduling framework.

Load balancing manages the jobs across the cores, and local scheduling directs the core to switch between threads. Thread switching (or context switching), say from T_i to T_j, requires saving the context of T_i and loading or restoring the context of T_j. Also, certain tasks must be performed when a thread completes its execution. In the simulation context, these tasks are basically updating the simulation system state. In our framework, we provide generic routines to do these tasks.

In essence, implementing the routines of load balancing and local scheduling are the programming effort needed to use our simulator to study the performance of a new scheduling algorithm. Again, to reduce the effort of implementing the scheduler further, we provide a default load balancing routine and sample local scheduling algorithm routines. These routines can be suitably modified to implement the new scheduling algorithm, unless the new scheduling algorithm is completely novel and does not follow the structure of load balancer and local scheduler combination. Even in that case, the entire scheduling can be designed from scratch with little effort to use our framework. In any case, from our experience, we are confident that, once the logic

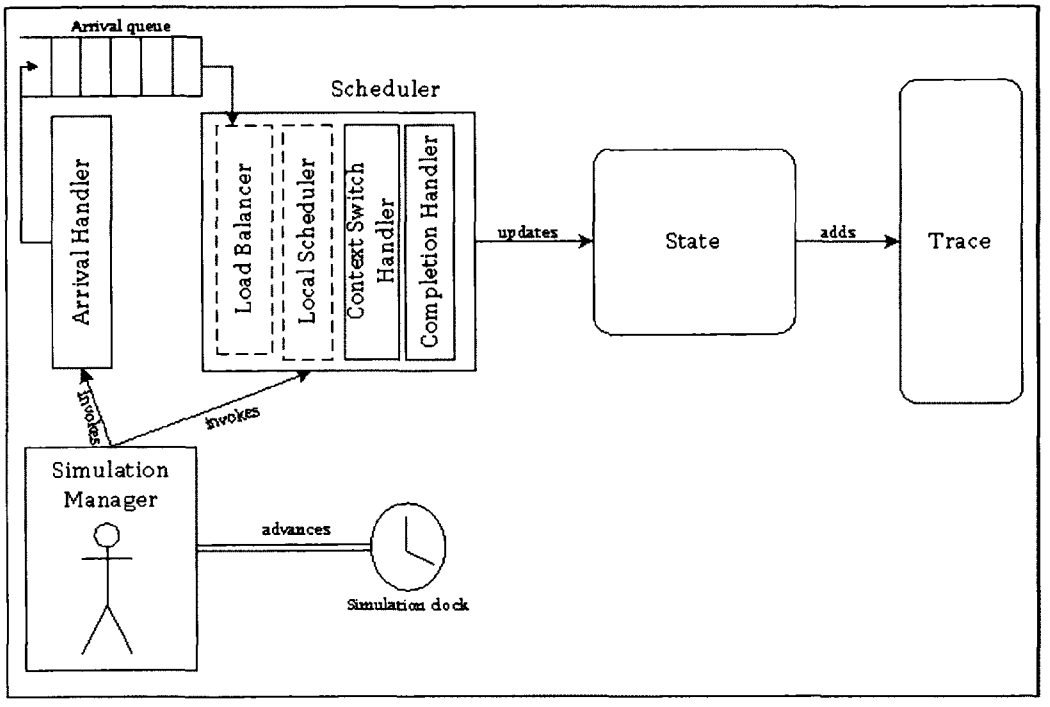

Figure 4.1: The Scheduler

of the new scheduling algorithm is completely understood, then implementing it in our simulator is straightforward. With this background on multicore scheduler, next we explain the overall multicore scheduler simulator, which is a part of the framework.

The simulator is illustrated in Fig. 4.1. It has seven components: simulation clock, simulation manager, arrival handler, arrival queue, scheduler, state, and trace. The scheduler has four routines: load balancer, local scheduler, context switch handler, and completion handler.

The simulation of the system basically involves updating the state of the system at every simulation time point, and incrementing the simulation clock. Simulation clock advancement and state update could be done in an integrated fashion. But, for the modular design of the scheduling framework, we decided to keep these two tasks separate.

In our framework, we designed a component called Simulation Manager to do the task of advancing the simulation clock and initiating the appropriate routines to update the system state. Updating the simulation system state typically involves recording the arrival of new jobs, and updating the state related to scheduling. To implement the arrival of new jobs, we introduced a queue called *arrival queue*, and implemented a routine to register the newly arrived jobs in the arrival queue. Updating the state related to scheduling is dependent on the scheduler logic, and it must be done as part of the implementation of the scheduler.

The simulation of the executions of jobs are recorded as simulation trace, and it is recorded at every scheduling point. To make this task generic, we have standardized the format of the execution trace and implemented a routine to add the trace appropriately during the simulation. Actually, this task of updating the trace is taken care of automatically by the context switch handler routine. With this background, we now introduce the architecture of the multicore scheduler simulation framework.

4.3 Architecture of MSS Framework

First we introduce some basic terminology used in our framework. Hereafter, to avoid confusion of what really refers to processor[1] in the simulation context, we avoid its usage in the rest of the thesis.

4.3.1 Terminology

- *Core:* The hardware execution unit.
- *Chip:* An integrated circuit containing one or more cores.

[1]Before multicore era, the term processor was used to refer to as an execution unit. Therefore, it was used synonymous with central processing unit (CPU). Now, with multicore technology, a processor has several execution units.

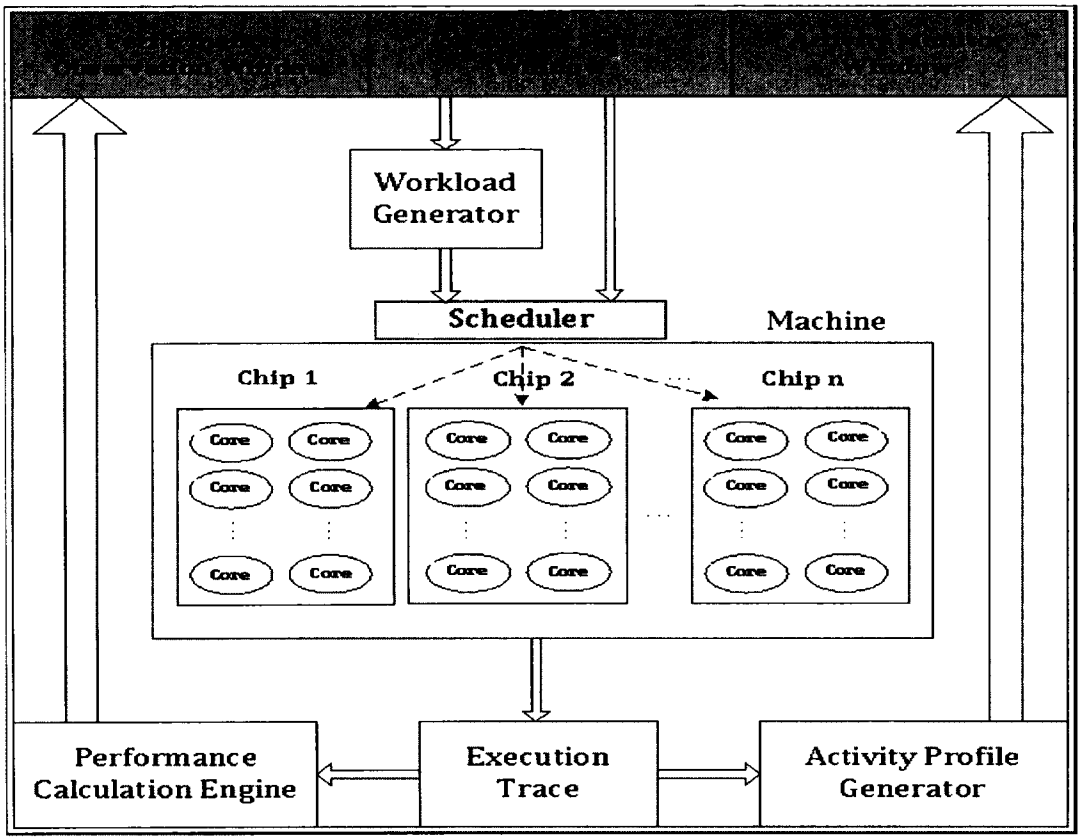

Figure 4.2: Architecture of MSS Framework

- *Machine:* A collection of chips designed to work together.
- *Thread:* The smallest unit that can be scheduled to a core for execution.
- *Gang:* A set of parallel threads that can be executed to achieve a task.

A higher level architecture of MSS framework is given in Fig. 4.2. The framework has five main logical components: workload generator, multicore machine, multicore scheduler, execution trace, and performance calculation engine. We explain them next.

4.3.2 Workload Generator

We have implemented the workload generator to generate two types of workloads: threads and gangs. Threads are generated for traditional applications and gangs are generated for parallel applications. The gangs of parallel threads typically execute in a synchronized manner.

The input to generate a workload of threads are: the numbers of threads, mean arrival rate, and mean service rate. To generate the workload of gangs, instead of the number of threads, the number of gangs is given. The number of threads within each gang is generated uniformly between the range 2 to M, where M is the number of cores.

The output will be a set of threads or gangs depending on the generator chosen. Each thread will have a unique id, arrival time, execution time, I/O points, priority, and application class. Similarly, each gang will have a gang id, arrival time, execution time, and a set of parallel threads with their own ids.

The I/O points of a thread are the times when the thread will go for I/Os. For example, assume that a thread has an execution time 50, and will go for I/O at time 5, 20, 30. In this case, I/O requests will be invoked after the thread is executed for 5 units, 20 units, and 30 units. The number of I/O points are generated uniformly within the execution time range, and the I/O wait time is generated uniformly within a range specified in the configuration.

4.3.3 Multicore Machine

The Multicore machine is the computing unit that is organized in a hierarchy, starting with machine at the highest level, as shown in Fig. 4.3.

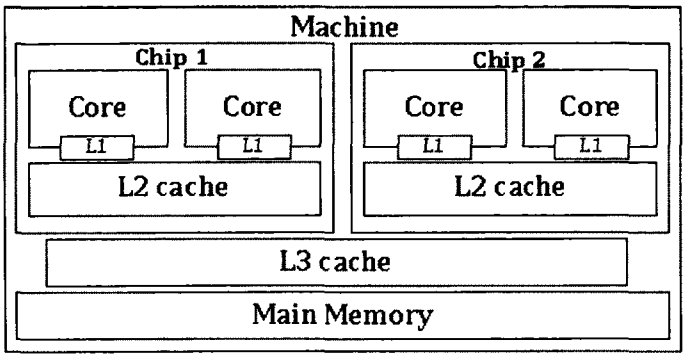

Figure 4.3: A Multicore Machine

A multicore machine contains one or more chips, and each chip contains two or more cores. The core is the physical execution unit. In practice, a core is capable of executing one or more threads in an interleaved way, referred to as hyper-threading. In our simulator we model a core to execute one thread at a time. We believe that, with a suitable scheduling policy, the performance of hyper-threading effect can be approximately simulated using single threaded cores.

A multicore machine has hierarchy of cache memories. Current multicore systems have three levels of cache memories, referred to as L1, L2, and L3. L1 is core level, L2 is chip level, and L3 is machine level. As shown in Fig. 4.3, L1 is local to each core, cores in a chip share L2 of that chip, and L3 is shared by all the cores in the machine.

4.3.4 Multicore Scheduler

As explained earlier, our framework implements a multicore scheduler having two logical components - load balancer and local scheduler. The load balancer is responsible for maintaining a desired balance of the system load. This task involves dispatching the new jobs to the appropriate local scheduler, and migrating jobs from one local scheduler to another when necessary. The local scheduler is responsible

```
<cid, tid, Ex-start, Ex-end, S>
<0,0,4,12,0>
<1,1,9,13,0>
<1,1,13,17,2>
<3,4,16,19,2>
<0,2,12,22,0>
.
.
.
<4,354,12459,12459,3>
```

Figure 4.4: Sample Execution Trace

for scheduling jobs to the cores for execution. Generally, each core is assigned a local scheduler, but other choices are possible. In our simulation, local scheduling is implemented to have centralized control.

In simulation context, the local scheduler primarily makes a decision to choose a job for execution, determines the amount of execution time, calculates its progress rate, and produces the trace. The progress rate of a job is the crucial design factor affecting the accuracy of execution, and it is dependent on several factors such as execution speed of the core and the contention for shared resources. We have used a simple cache contention model, but it can be easily replaced with an implementation of a more refined model.

4.3.5 Execution Trace

During the simulation, the execution trace is recorded at every context switch to generate the activity profile and to compute the performance metrics. There are two types of traces collected: execution trace and I/O trace. The execution trace is a collection of quintuple, as shown in Fig. 4.4.

In the execution trace, each quintuple has a core id (cid), thread id (tid), execution start time (Ex-start), scheduling end time (Ex-end), and a status (S). Status is 0 if preempted by quanta expiration, 1 if preempted by higher priority thread, 2 if going

```
<tid, I/O-start, I/O-end>
<4,19,49>
<4,52,69>
<19,52,76>
<13,43,83>
<34,85,87>
.
.
.
<681,12045,12070>
```

Figure 4.5: Sample I/O Trace

for I/O, and 3 if completed. The I/O trace is a collection of triples, as shown in Fig. 4.5, each triple has thread id, I/O start time (I/O-start), and I/O end time (I/O-end).

4.3.6 Performance Calculation Engine

The performance study primarily aims to determine how well the algorithm responds to satisfy certain criteria such as response time, fairness, and utilization of resources. The performance calculation engine calculates these values for a given set of data, and the result can be passed to the performance observation window for the users to study. The performance criteria widely used to study scheduling algorithms are the five metrics: throughput, CPU utilization, turnaround time, waiting time, and response time [37, 38].

In addition, we have included three more measures: (i) interactive response time, (ii) bypass count, and (iii) slow down factor. Interactive response time is introduced to observe the interactive response of tasks. It is defined as the time from when a thread is ready for execution to the subsequent start of execution. To avoid starvation, we introduce the metric of bypass count. Bypass count will indicate the number of threads which bypassed a waiting thread in scheduling, and it is an indication of unfair scheduling. We use a bypass count graph to see the level of fairness that a scheduling algorithm can assure. Finally, a relative performance would be interesting

for resource allocation purpose. For that, we include slow down factor.

The following list of performance metrics is supported in the proposed framework.

1. *Throughput:* The total number of jobs completed execution in one unit of time. Assuming the simulation starts at time 0, and n jobs complete in T_s period, the throughput TP is computed as follows:

$$TP = \frac{T_s}{n} \tag{4.1}$$

2. *Core utilization:* This is the percentage of time the core spends on executing jobs. Let T_s be the total simulation time and T_b be the amount of time the core is busy. Then, the utilization (U_c) of core c is computed as follows:

$$U_c = \frac{T_b}{T_s} \times 100\% \tag{4.2}$$

3. *Core idle time:* This is the percentage of time the core is idle without jobs to execute. The idle time ($Idle_c$) of core c is computed as follows:

$$Idle_c = (1 - \frac{T_b}{T_s}) \times 100\% \tag{4.3}$$

4. *Core wait time:* The total time that the thread waits for core.

5. *I/O Wait time:* The total time taken for all I/O waits of the thread.

6. *Wait time:* The sum of core wait time and I/O wait time of the thread.

7. *Turnaround time:* The sum of wait time and execution time of the thread.

8. *First response time:* The time from submission to start of execution of the thread.

9. *Interactive response time:* The time from when the thread is ready for execution to the subsequent start of execution.

10. *Slow down factor:* Slow down factor of a thread is defined as the ratio between its turnaround time and execution time. This metric is used to measure the delay suffered by a job against its actual execution time. If wt_i and ex_i, respectively, are the wait and execution times of i, then the slowdown factor S_i of the thread i is computed as follows:

$$S_i = \frac{wt_i}{ex_i} \tag{4.4}$$

11. *Bypass Count Graph:* Every thread is associated with a bypass counter. This counter is incremented whenever it is bypassed another thread in scheduling. The bypass count graph reflects how many threads have bypassed a waiting thread in scheduling. It gives the sense of fairness that the scheduling algorithm can assure.

Let $x_1, x_2, ..., x_n$ be the values, the average $\overline{x}$ and standard deviation σ are computed as follows:

$$\overline{x} = \frac{\sum_{j=1}^{n} x_j}{n} \tag{4.5}$$

and

$$\sigma = \sqrt{\frac{\sum_{j=1}^{n} (x_j - \overline{x})^2}{n}} \tag{4.6}$$

4.3.7 Statistical Measures

For these metrics, when applicable, we calculated minimum, maximum, average, and standard deviation. Traditionally, average and percentage are used to study the performance in this context. That is, typically, throughput and utilization are computed in percentage, and average is computed for turnaround time, waiting time, and response time. We believe better measures than average must be used in analysing these metrics.

As cloud computing is fundamentally a service oriented system, its primary goal is to provide quality service to its customers. Although the term quality of service is often used and directly related with response time, it is not well interpreted in the context of computing and communication systems. As indicated in [61], perceived quality of customers need not be directly related with minimal response time. The study indicates that users are often unaware of the quality differences until it crosses certain threshold. Therefore, quality of service need not always be related to the widely used metrics such as minimal response time or minimal average response time. Thus, we believe that a predictable response time is a more appropriate measure for the quality of service in the cloud computing context than other measures such as average response time and resource utilization.

One such measure we discussed earlier is predictable response time. Predictable response could be measured using variance or standard deviation. Therefore, we decided to include the metric of standard deviation (i.e., square root of variance) in our framework. We believe that standard deviation is a better measure of predictability than average value.

4.4 Activity Profile Generator

Activity monitor provides visualization of how the algorithm schedules the threads among the cores. Activity profile generator is responsible for providing the data for the visualization. It basically derives the data from the execution trace. From the trace, it generates data for every clock tick. The data contains the core status as *idle* or *occupied*. Also, if occupied, the data contains the information about the thread and its status, whether it is a newly arrived thread or preempted thread or migrated thread from a different core. Activity profile generator differentiates these different states of the core by assigns different colors. The activity monitor window visualizes the derived data of core to thread allocation. Using this component, we can visually observe core utilization, load balancing, and the individual core statistics.

4.5 User Interface

The multicore scheduling simulator has three main user windows, and the interface within each window is organized as hierarchical panels. We explain these windows next.

4.5.1 Performance Parameter Setting Window

Performance parameter setting window is used to configure the parameters for the simulation. It consists of two screens. First screen, shown in Fig. 4.6, is used to set the workload generation parameters. The input for the workload generator are the number of jobs, arrival time distribution, mean arrival rate, and job execution time range.

The parameter setting window has the provision to create more than one work-

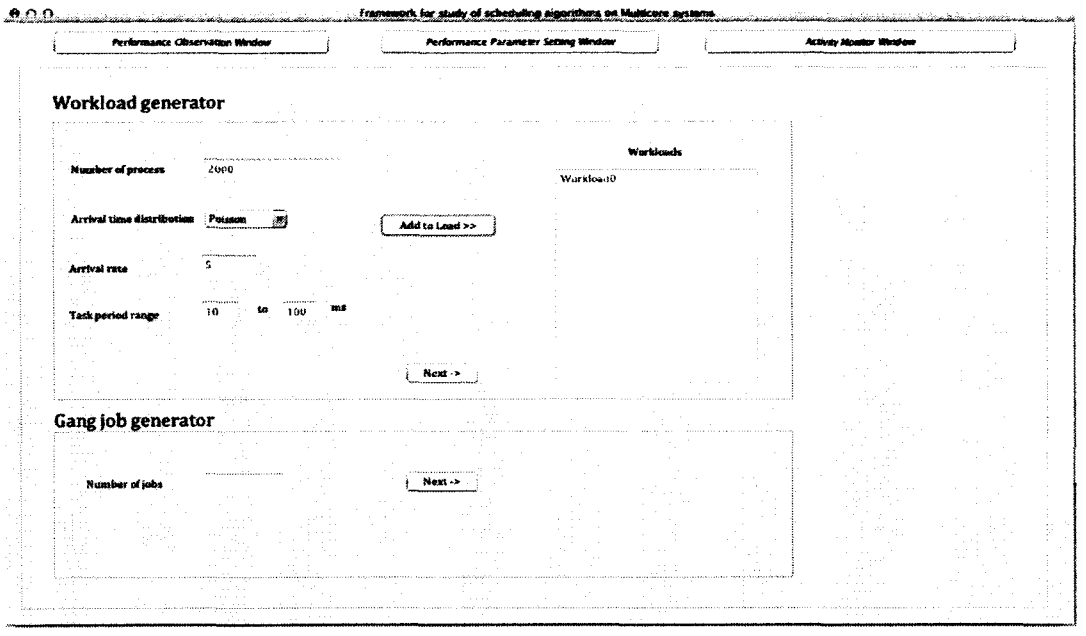

Figure 4.6: Parameter Setting Window

load. The window takes number of process, arrival rate distribution, arrival rate, and execution range as the parameters for each workload. Once you configure the above mentioned parameters, you can add it to the workload list by pressing the 'Add to Load' button which will add to the list of workloads and will display it in the right side of the window. Using this option, we can configure more than one workload.

The simulations run window, shown in Fig. 4.7, is designed to get input for creating simulation runs. Using this window, several simulation runs can be configured. The parameters required for each run are the number of cores, workload selection, and scheduling algorithm. Using the simulation run window, we will be able to create simulation runs with different combinations are listed below:

1. number of cores vs workload with the same scheduling policy

2. number of cores vs scheduling policy under the same workload

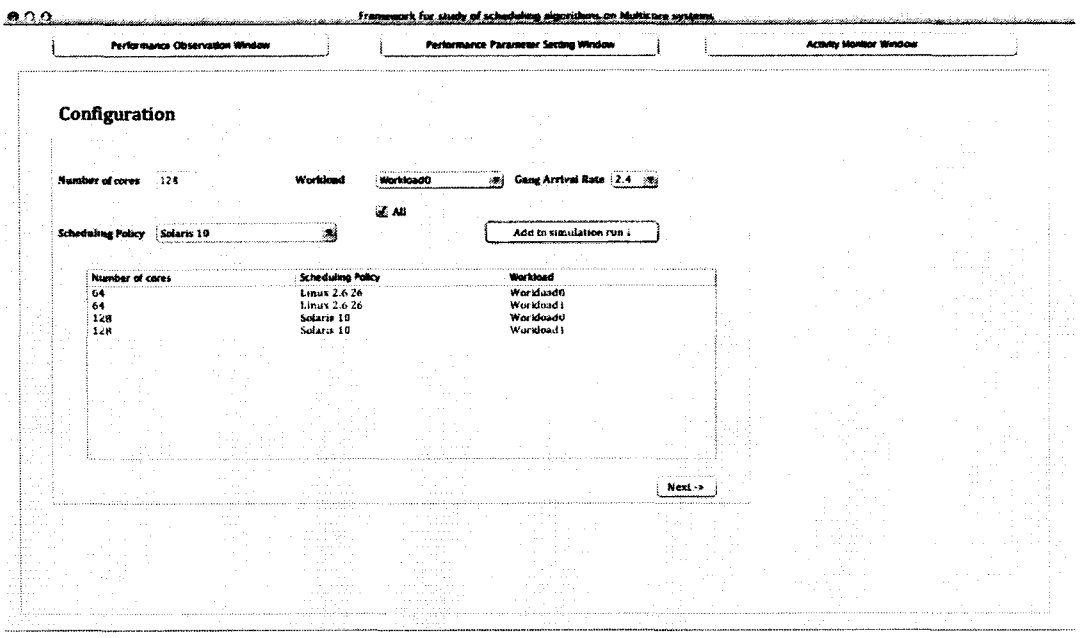

Figure 4.7: Simulations Run Window

3. workload vs scheduling policy when executing with same number of cores

This simulations run setting feature simplifies the effort involved in simulating the scheduling algorithms under various conditions.

4.5.2 Performance Observation Window

Performance observation window, shown in Fig. 4.8, offers various performance metrics that can be represented in charts. The window has a list of performance metrics which can be chosen to see the computed values. Additionally, there is an option to choose two different simulation run and compare the results. The performance metrics can be analysed by varying the number of cores and arrival rate.

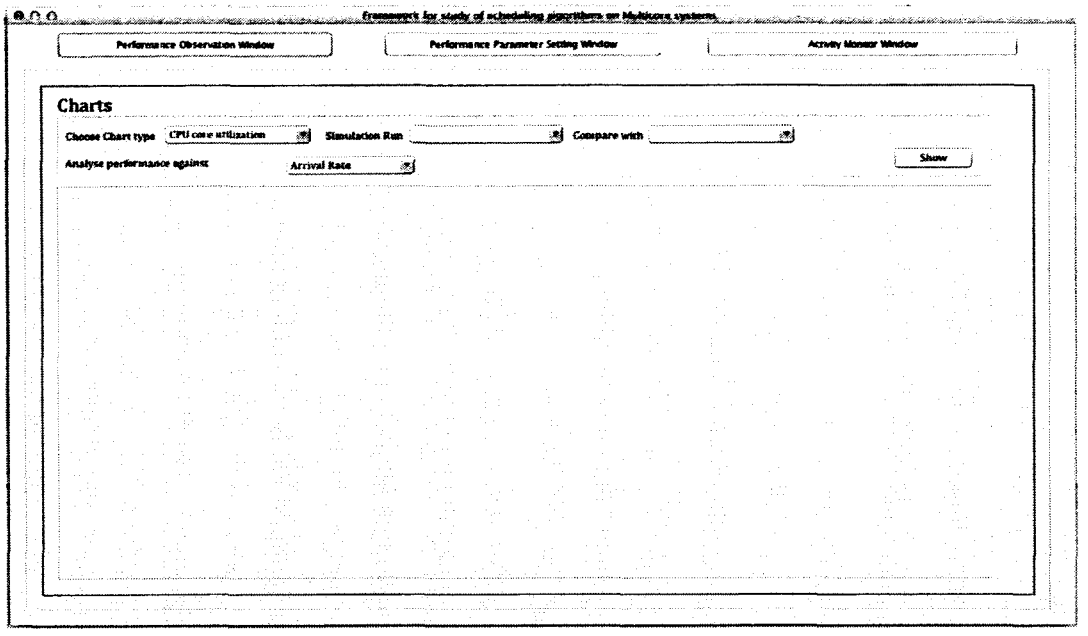

Figure 4.8: Performance Observation Window

4.5.3 Activity Monitor Window

Activity monitor window is an interesting component in this simulator. This screen shows the core usage in a graphical representation. With this screen, shown in Fig. 4.9, we can to analyse the following:

1. Core utilization
2. Thread migration
3. Execution thread list

Using Activity monitor window, we can visualizes how the threads are distributed among the cores and how effectively the load is balanced. To see the individual core performance, a core monitor child window is attached to each core to show its statistics. The child window is shown in Fig. 4.10

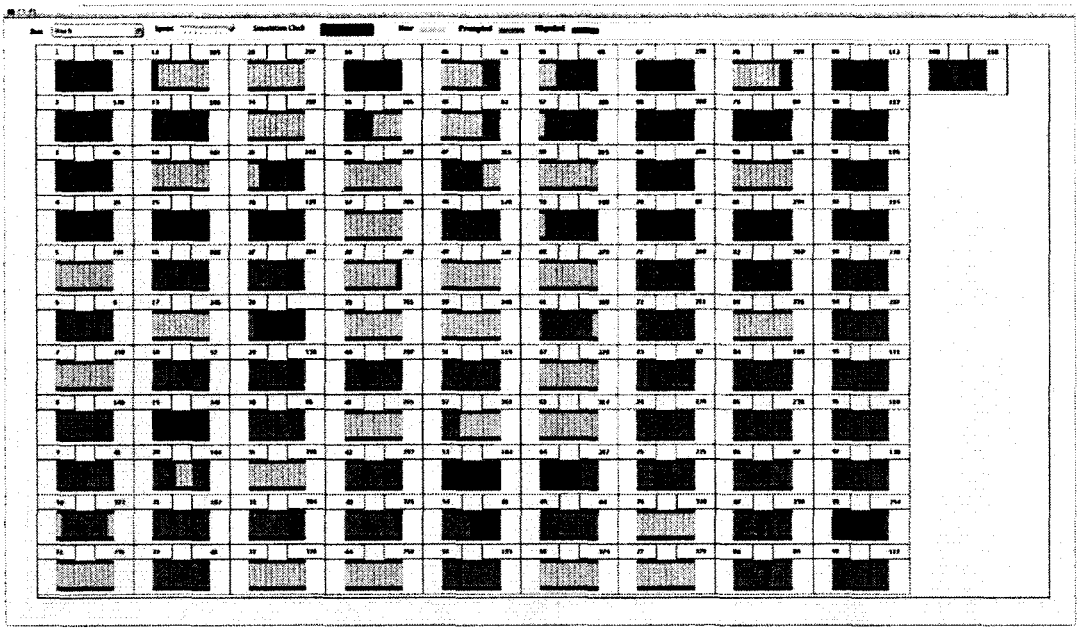

Figure 4.9: Activity Monitor Window

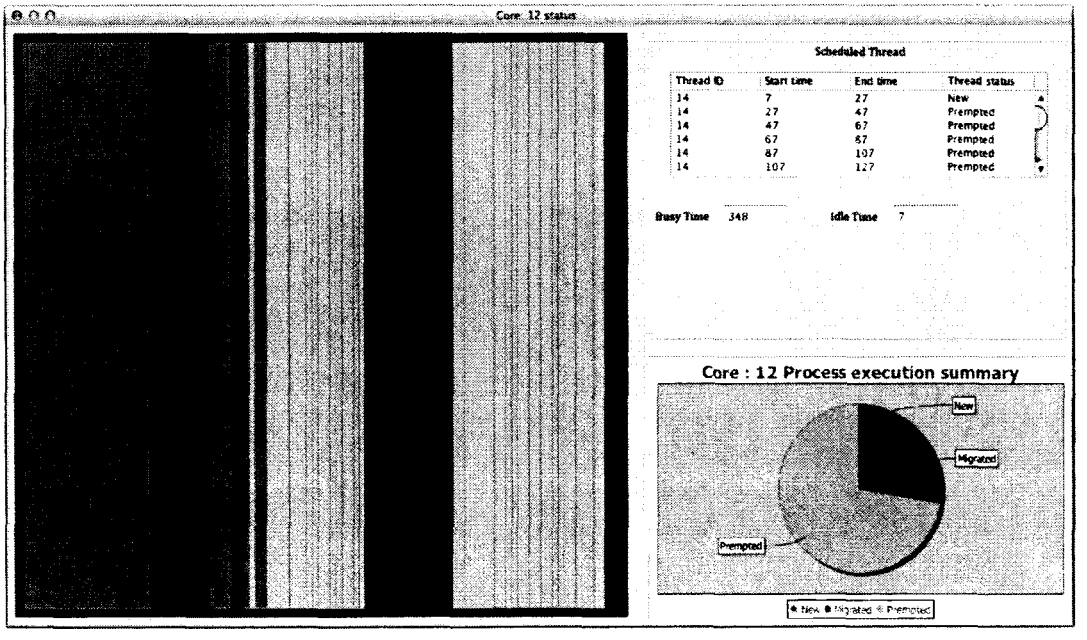

Figure 4.10: Core Monitor Window

The core monitor window has three parts. The first part (left part) is to display the threads running in that core using different colors. The second part (right upper part) is to display the statistics of busy and idle time of the core. A thread scheduled in that core could be new, preempted, or migrated from another core. The ratio of these three types of threads scheduled in that core is displayed in the third part (right lower part) of the window. This part shows how many threads are migrated from other cores due to load balancing. Such visual analysis is sometimes useful to capture unusual behavior and patterns.

4.6 Summary

In this chapter, we presented a new framework for simulation of multicore scheduling algorithms. The framework is flexible, and serves as a base for designing new scheduling algorithms and conducting experimental study on existing scheduling algorithms for multicore processors.

Chapter 5

Case Studies - Linux and Solaris Scheduling Algorithms

To test and illustrate the functionality of the proposed simulator, we implemented the recent versions of Linux and Solaris 10 scheduling algorithms. We first present the algorithms, and then describe the simulation experiments and observations. The recent version of Linux scheduler is called completely fair scheduler (CFS)[1], and the Solaris scheduler is referred to as kernel dispatcher.

Although Linux and Solaris are popularly used operating systems, the documentations precisely describing the scheduling algorithms are rarely published. We had to reconstruct the algorithms using information obtained from different sources.

Both schedulers have two logical components - a load balancer and a local scheduler. The load balancer is responsible for maintaining a desired balance of the system load. This task involves dispatching new threads to the appropriate local scheduler, and migrating threads from one local scheduler to another when necessary. The local

[1]CFS has been implemented starting from Linux 2.6.23.

scheduler is responsible for scheduling threads to the cores for execution. We first present the load balancer of these two algorithms together first, and then the local scheduling algorithms separately.

5.1 Load Balancing

Load balancing involves four factors: (i) initial placement; (ii) migration criteria; (iii) migration policy; and (iv) frequency of balancing.

- **Initial Thread Placement** - Initial thread placement in both Linux and Solaris is the same: the new threads are dispatched to the lightly loaded cores.
- **Load Balancing Criteria** - The difference in the number of threads between any two cores is less than one.
- **Migration Policy** - Linux migrates threads from heavily loaded cores to lightly loaded cores to satisfy the load balancing criteria. Solaris, when the choice occurs, moves the thread to the core in different chip than to the core in the same chip, to reduce the cache conflict.
- **Balancing frequency** - Load balancing in Linux is done every 200ms, and the load balancing in Solaris is done every 100ms.

Next, we present the local scheduling algorithms of Linux and Solaris. The threads in both algorithms are classified as real-time[2] thread and normal thread. Here, real-time implies that these threads have higher priorities than the normal threads, and therefore real-time threads are always executed before normal threads. We first present how the real-time threads are handled in both Linux and Solaris.

[2]The term real-time in this context is misnomer that it does not associate any specific deadline to meet, and no traditional scheduling algorithms like rate monotonic (RM) or earliest deadline first (EDF) have been used to schedule these threads.

5.1.1 Real Time Scheduler

Whenever a real-time thread arrives, it preempts the normal thread and the real-time thread is scheduled to run. When a real-time thread is executing, if another real-time thread with higher priority arrives, then the current thread is preempted and the higher priority thread is scheduled. Within the same priority level of real-time threads, Linux uses either the First-In-First-Out (FIFO) or the Round Robin (RR) scheduling policy, and Solaris uses FIFO.

5.2 Linux Scheduler - Completely Fair Scheduler (CFS)

The basic idea behind CFS is to ensure fair share among threads in the overall execution. This is achieved by quanta allocation. The execution time plays a key role in quanta computation. CFS maintains the amount of time that a thread has utilized the core before, referred to as *vruntime*. The thread with the smallest *vruntime* has the highest preference to be selected next for execution. Calculation of quanta and *vruntime* is given later.

Instead of run queue, for efficiency, CFS maintains a data structure called Red-Black tree, sorted by *vruntime* key. The scheduler picks from the left-most child of the tree which is the smallest *vruntime* thread for execution. The pseudocode of CFS algorithm is given in Table. 5.1.

5.2.1 Calculation of Quanta

The quanta value calculation plays a key role in CFS. The time quanta $Q[T_i]$ of a thread T_i is calculated using the following formula:

Data Structures: Red-Black tree - RBT, Thread T_i
1. **while** ($RBT \neq$ empty) **do**
2. select leftmost thread T_i from RBT
3. compute time quanta
4. schedule T_i
5. **end while**

Table 5.1: Completely Fair Scheduling Algorithm

$$Q[T_i] = \frac{T_i.weight}{\sum_{j \in RBT} T_j.weight} \times P \tag{5.1}$$

where,

- $T_i.weight$ is the weight value corresponds to nice [3] value of T_i. (Every nice value is mapped to a weight value.)

- $$P = \begin{cases} sched\_latency & \text{if } n > nr\_latency \\ min\_granularity \times n & \text{otherwise} \end{cases} \tag{5.2}$$

  where n is the number of threads in RBT, and $sched\_latency$, $nr\_latency$, and $min\_granularity$ are constants. In the current implementation, these values, respectively are 6, 8 and 0.75 [63]. The details of how these values are determined and their significance are not known.

[3] In Linux, the priority of the thread is controlled with the nice value. Nice value ranges between -20 and 19. Lower value corresponds to higher priority. The user level command nice can be used to lower the priority of a thread (i.e., to be nice to other users).

5.2.2 Calculation of *vruntime*

For every clock tick, the scheduler calculates *vruntime* of the executing thread and also decreases its time quanta. Preemption of a thread occurs when quanta expires or RBT has a thread with smaller *vruntime*. The virtual *vruntime* of a thread T_i is computed as follows:

$$T_i.vruntime = \frac{weight_0}{T_i.weight} \times T_i.runtime \tag{5.3}$$

where, $weight_0$ is the value corresponding to the nice value of 0 and $T_i.runtime$ is the execution time consumed so far by the thread T_i.

5.3 Solaris Scheduler

Solaris 10 kernel dispatcher does both load balancing and local scheduling. Solaris has implemented two schedulers - fair share scheduler and a default scheduler. The fair share scheduler is typically used in server environments. We have implemented the default scheduler, that we will describe next.

For simplicity, in Solaris, we consider just threads are scheduled for execution[4]. Solaris maintains the priority range of 0 - 169. Priority range of 160-169 is reserved for Interrupts, and the rest of the priorities are assigned to different scheduling classes (see Appendix).

Solaris manages the threads in the following six different scheduling classes.

[4]In actual implementation, each application process in Solaris may contain one or more application threads and each application thread is scheduled (mapped) to a virtual core called light weight process (LWP). Each LWP is implemented using a kernel thread, and these kernel threads are eventually scheduled and executed by the physical core.

1. *Time share (TS)*

2. *Interactive (IA)*

3. *Fair share scheduling (FSS)*

4. *Fixed priority(FX)*

5. *Real Time (RT)*

6. *System (SYS)*

By default, threads created by the window manager are assigned to IA class for better interactivity, and the rest are assigned to TS class. System threads are created by the operating system. Other threads are created using different levels of system privileges. FSS class is used when the fair share scheduler is invoked.

Priority of a thread may be specified or inherited from the parent. In fixed priority class, threads have the same priority throughout their execution. RT class is the highest priority thread class which requires attention right away, and needs to be scheduled immediately. Next to the threads in real time class, the kernel threads get attention. Finally, the threads in the classes TA, IA, and FX are scheduled. The scheduler always chooses the highest priority thread for execution. The scheduling classes priority ranges are summarised in Table 7.1 (see Appendix).

5.3.1 The Default Solaris Scheduler

In Solaris, except RT, every scheduling class has a local scheduling queue for every core. For RT class, a global level kernel preemption queue is maintained for every chip.

After choosing a thread for scheduling, the next task is to obtain the quanta value. For that, Solaris maintains a set of tables, called dispatch tables (see Appendix), from which the quanta value is obtained. Also, the scheduler provides the fairness among the threads by boosting the priority up/down, in response to the following events.

- A thread successfully completes its execution for specified time quanta. Here, the thread priority has to be boosted down to give fairness to other threads.
- A thread comes back from an I/O wait. Here, its priority has to be boosted up, so that it will execute soon.
- A thread is waiting in its ready queue beyond certain threshold time period. Here, the priority has to be boosted up to give chance to execute soon.

These scheduling subtasks are performed by specific routines called tick processing and update processing (see Appendix). Tick processing is responsible for managing quanta and invoking preemption. Update processing is responsible for boosting the priority up/down. Dispatch tables are used to obtain new quanta, waiting threshold, and new priority.

With this set up, the scheduling policy is: at any scheduling time point, choose the highest priority thread in the system and schedule for execution. If a higher priority thread arrives when a lower priority thread is executing, then it will be preempted to allow the higher priority to execute. A higher level description of Solaris scheduling algorithm is given in Table 5.2.

Data Structures: Scheduling queues - DQ_TS, DQ_IA, DQ_FX, KPQ_RT, Thread T_i

1. **while** ($KPQ\_RT \vee DQ\_TS \vee DQ\_IA \vee DQ\_FX \neq$ empty) **do**
2. **if** ($KPQ\_RT \neq$ empty)
3. select highest priority thread T_i from KPQ_RT
4. get the time quantum $Q[T_i]$ from dispatch table
5. schedule T_i
6. **else**
7. pick highest priority thread T_i from DQ_TS, DQ_IA, DQ_FX
8. get the time quantum $Q[T_i]$ from dispatch table
9. schedule T_i
10. **end if**
11. **end while**

Table 5.2: Solaris 10 Scheduling Algorithm

5.4 Simulation Experiments

To illustrate the functionality and use of the proposed simulator, we conducted two sets of experiments:

1. **Experiment 1:** In this experiment, the workload is fixed and the number of cores is varied. The simulation parameters used for this experiment are given in Table 5.3.

Parameter	**Value**
Number of Threads	5000
Mean Arrival rate[5]	2.5
Arrival distribution	Poisson
Execution time distribution	Exponential
Number of cores	10, 50, 100, 200, 500

Table 5.3: Simulation Parameters

[5]We use generic unit for arrival rate. The inter arrival times derived from the distribution are real numbers. They are then scaled to integer units of simulation clock. For example, the inter arrival time 0.4 is scaled to 4 simulation clock units.

2. **Experiment 2:** In this experiment, the number of cores is fixed and the workload is varied. The simulation parameters used for this experiment are given in Table 5.4.

Parameter	Value
Mean Arrival rate	2.5, 3.5, 4, 5
Time period	1 minute
Arrival distribution	Poisson
Execution time distribution	Exponential
Number of cores	50, 100, 150, 200

Table 5.4: Simulation Parameters

Linux schedules threads based on how much execution time it is consumed and Solaris does by how long a thread waits without execution. We believe scheduling thread based on wait time would give better response time and predictability than scheduling done based on execution time. Based on this observation, we make the following hypothesis.

Hypothesis: Solaris scheduler will have better and predictable response time than Linux scheduler.

Predictable response time is studied using standard deviation of response time. We computed the core utilization, turnaround time, standard deviation of turnaround time, and interactive response time and its standard deviation for both experiments and we explain our observations next.

5.4.1 Observations on Experiment 1

- *Observation on Core Utilization:* We observe that both algorithms keep the cores 99% busy. In terms of core utilization, there is no significant difference between these two scheduling algorithms. The consistent behavior is due to their load balancing which periodically runs to evenly distribute the workload

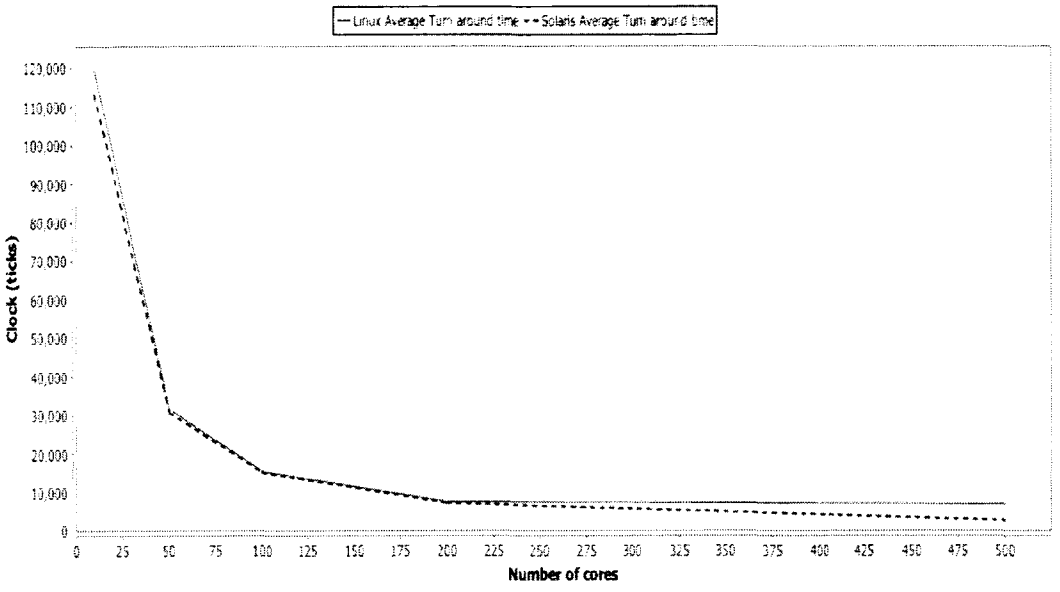

Figure 5.1: Average Turnaround time: Linux and Solaris Scheduling Algorithms (by varying the number of cores)

among the cores.

- *Observation on Average Turnaround Time:* The average turnaround time graphs of Linux and Solaris scheduling algorithms are shown in Fig. 5.1. The average turnaround time gradually decreases as the number of cores increases. Both Linux and Solaris algorithms show almost same behavior until 200 cores, and after 200 cores, Solaris performs better than Linux by a factor of 10 when the number of cores is 500. This is due to the fact that Solaris boosts up the thread's priority when the thread is waiting in the ready queue longer.

- *Observation on Standard deviation of Turnaround Time:* The standard deviation of turnaround time graphs of Linux and Solaris scheduling algorithms are shown in Fig. 5.2. We observe that Solaris and Linux show similar behavior between 10 and 200 cores. As we increase the number of cores, Solaris shows

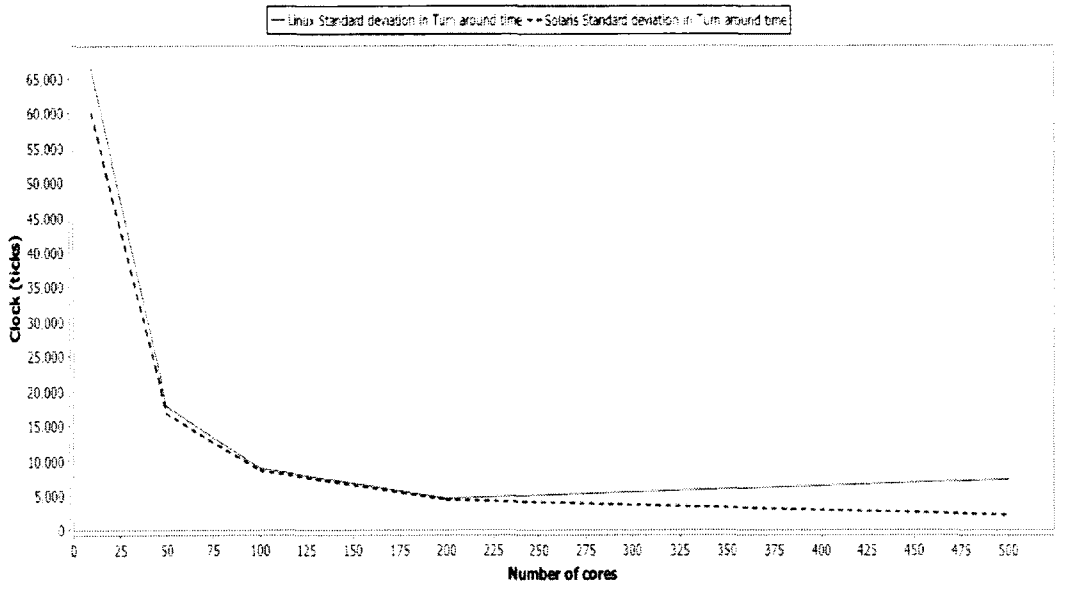

Figure 5.2: Standard deviation of Turnaround time: Linux and Solaris Scheduling Algorithms (by varying the number of cores)

better predictability than Linux.

- *Observation on Interactive Response Time:* The average interactive response time graphs of Linux and Solaris scheduling algorithms are shown in Fig.5.3. We observe that Solaris outperforms Linux consistently providing better interactive response time. This is because, whenever a thread stays in ready queue and reaches the maximum wait time, Solaris boosts up the thread priority. Also, whenever a thread returns from I/O, the thread priority is boosted up so that it can be scheduled soon.

- *Observation on Standard deviation of Interactive Response Time:* The standard deviation of interactive response time graphs of Linux and Solaris scheduling algorithms are given in Fig. 5.4. The prediction of how soon the thread will be scheduled for execution, when it is ready, is measured by computing the standard

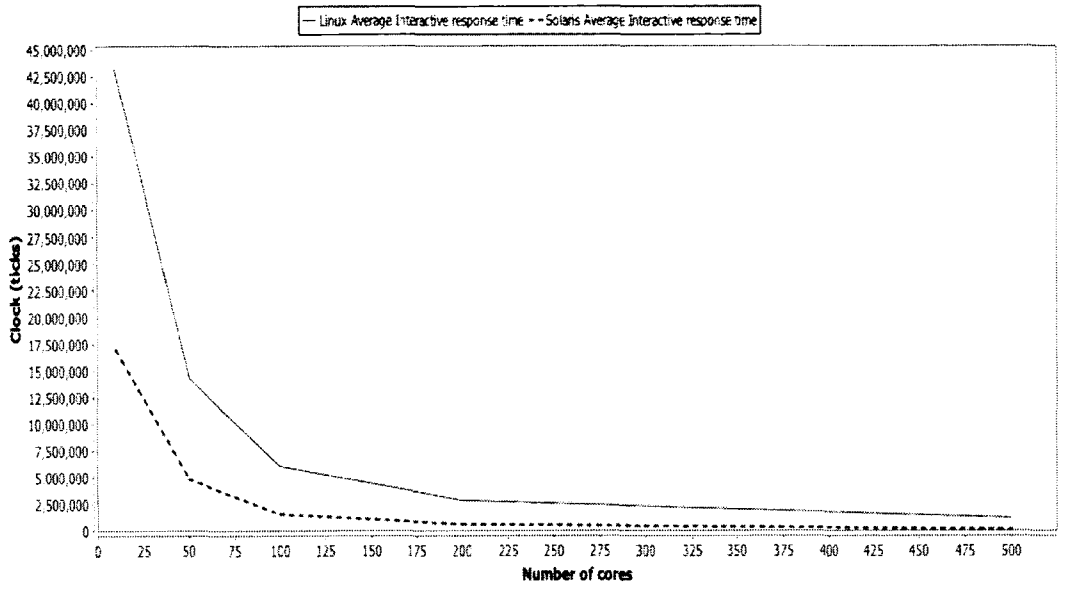

Figure 5.3: Average Interactive Response time : Linux and Solaris Scheduling Algorithms (by varying the number of cores)

deviation in interactive response time. It is clearly visible that Solaris performs better than Linux by assuring better fairness to the threads. As mentioned before, the boosting of the priority by Solaris influences the predictability better than Linux.

5.4.2 Observations on Experiment 2

- *Observation on Core Utilization:* We observed that both algorithms keep the cores busy and there is no significant difference.

- *Observation on Turnaround Time:* The average turnaround time graphs of Linux and Solaris scheduling algorithms are shown in Fig. 5.5. The average turnaround time constantly increases when the arrival rate increases. From these experiments, it is clearly seen that Solaris performs better then Linux,

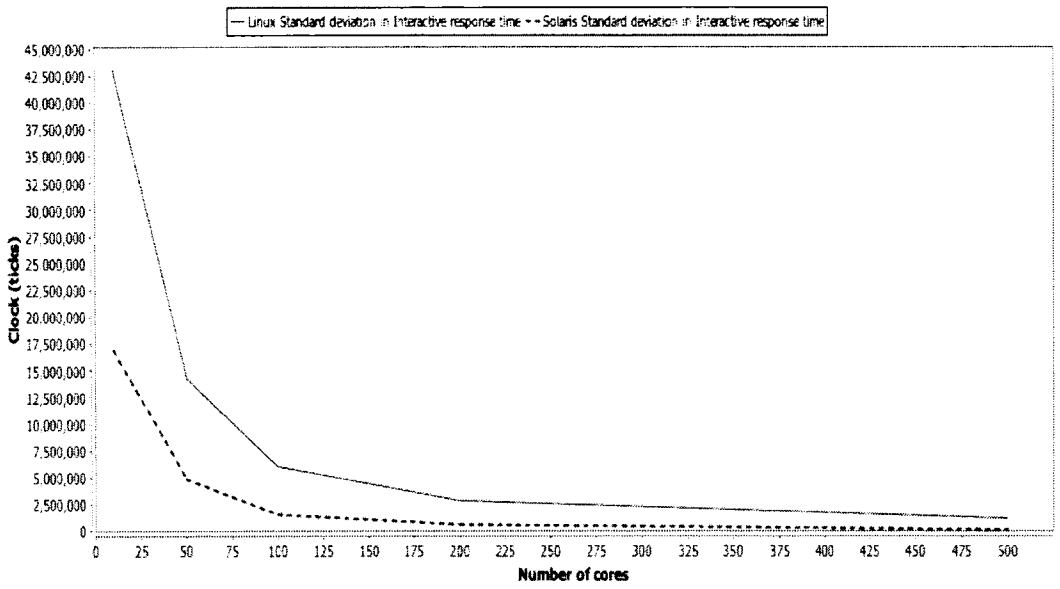

Figure 5.4: Standard deviation of Interactive Response time: Linux and Solaris Scheduling Algorithms (by varying the number of cores)

when the number of cores is 50, but the trend gradually converges as the number of cores increases.

- *Observation on Standard deviation in Turnaround Time:* The standard deviation of turnaround time graphs of Linux and Solaris scheduling algorithms are shown in Fig. 5.6. We observe that the predictability also increases when the arrival rate increases. Compared to Linux, Solaris predictability is always better.

- *Observation on Interactive Response Time:* The average interactive response time graphs of Linux and Solaris scheduling algorithms are shown in Fig. 5.7. Solaris outperforms Linux consistently providing low interactive response time. As explained earlier, the priority boosting of Solaris heavily impacts the interactive response time. This is an expected behavior.

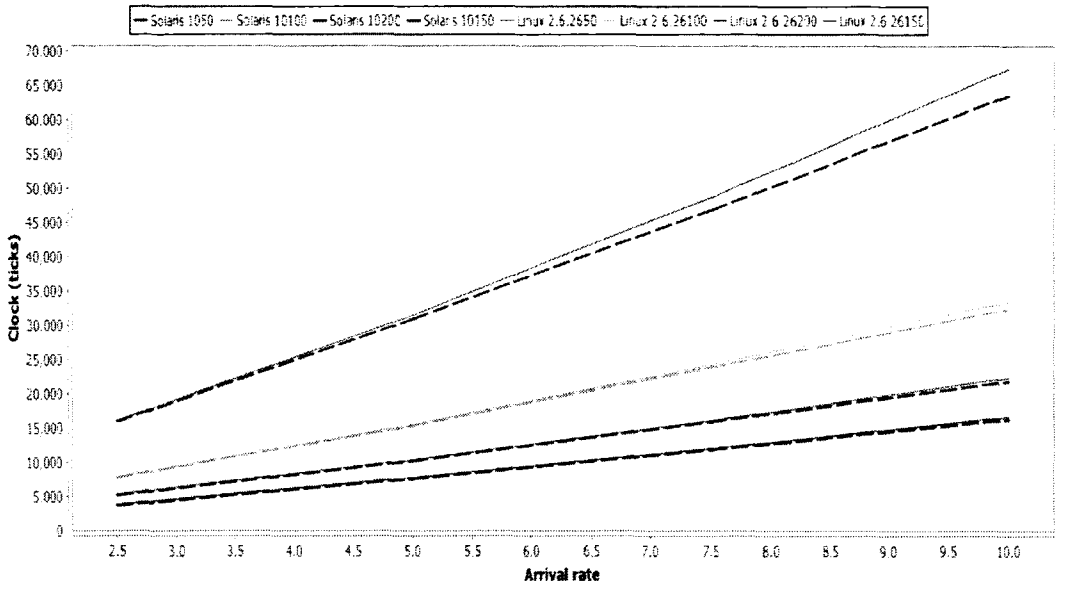

Figure 5.5: Average Turnaround time: Linux and Solaris Scheduling Algorithms (by varying the workload)

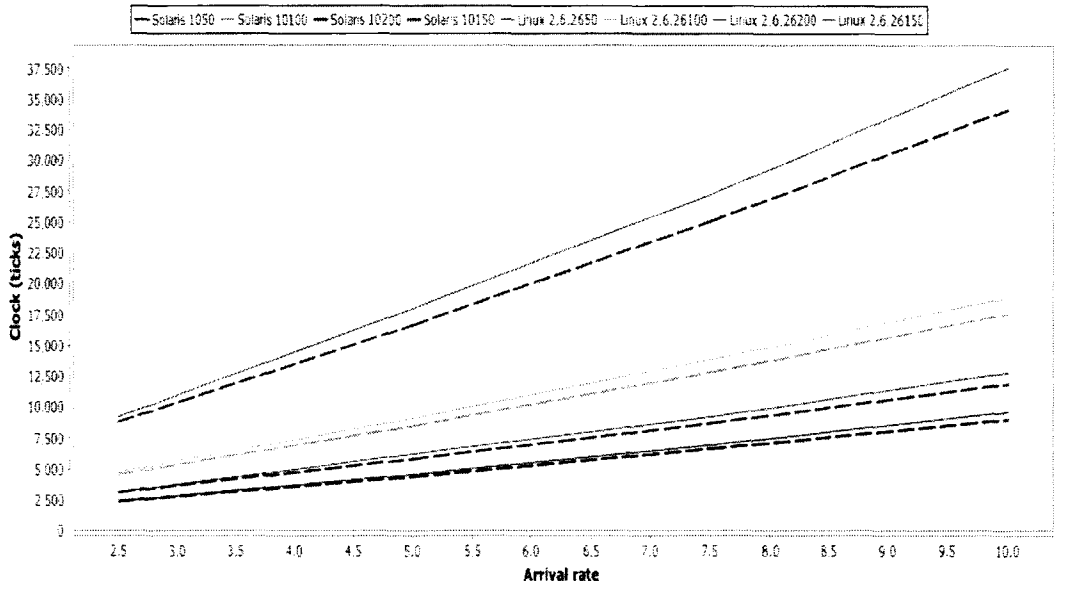

Figure 5.6: Standard deviation of Turnaround time: Linux and Solaris Scheduling Algorithms (by varying the workload)

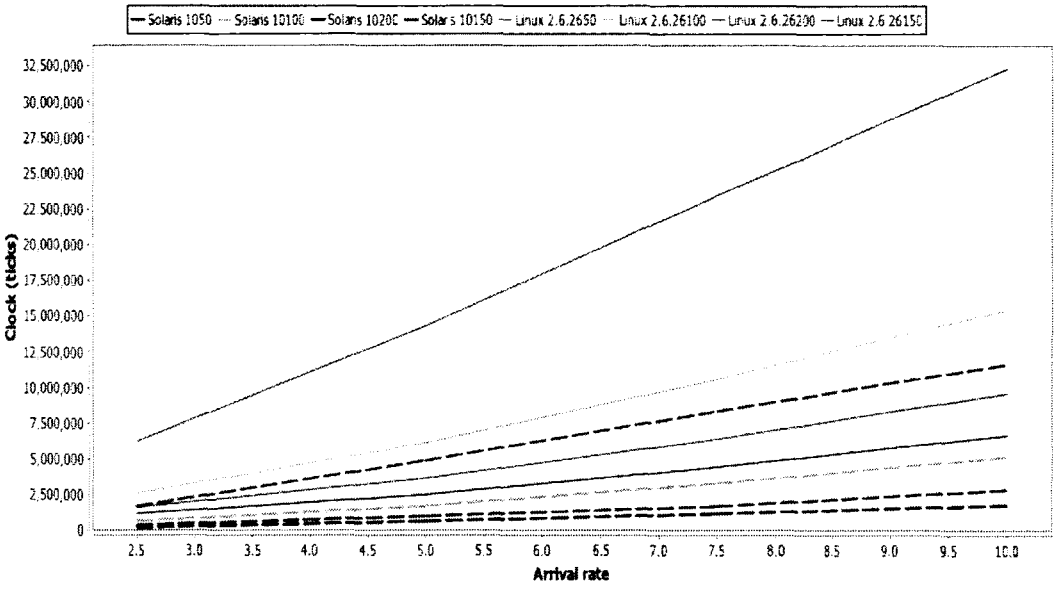

Figure 5.7: Average Interactive Response time: Linux and Solaris Scheduling Algorithms (by varying the workload)

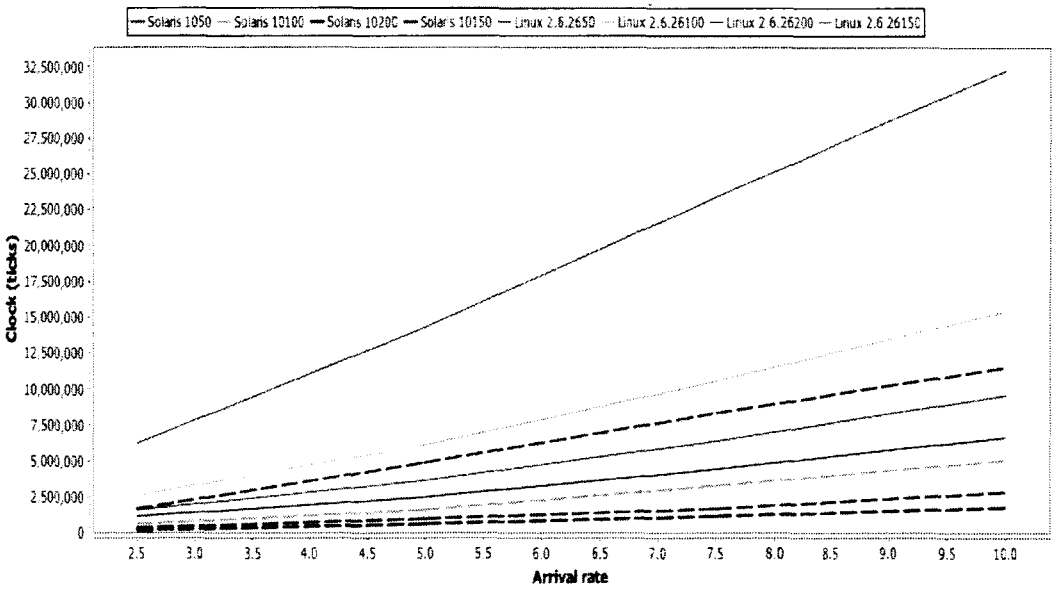

Figure 5.8: Standard deviation of Interactive Response time: Linux and Solaris Scheduling Algorithms (by varying the workload)

- *Observation on Standard deviation of Interactive Response Time:* The standard deviation of interactive response time graphs of Linux and Solaris scheduling algorithms are shown in Fig. 5.8. We observe that Solaris predictability is much better compared to Linux. The trend is consistent.

These observations confirm our hypothesis, which is an evidence that the implementation of the scheduler is fairly accurate.

5.5 Summary

In this chapter, we have presented the simulation experiments of Linux and Solaris scheduling algorithms. As we expected from our hypothesis, the simulation experiments prove that Solaris scheduler offers better and predictable response time than Linux scheduler. From these experiments, we are confident that our scheduler simulator implementation is fairly sound.

Chapter 6

A Fair and Efficient Gang Scheduling Algorithm

The trend in multicore processors indicates that all future processors will be multicore, and hence the future cloud systems are expected to have their nodes and clusters based on multicore processors. So the processor scheduling in the future systems will most likely be all multicore processor scheduling. Therefore, we believe, multicore scheduling is fundamental to future cloud computing performance. Also, due to multicore revolution, a considerable portion of large applications will be parallel programs. From the literature, we can see that gang scheduling is a dominant strategy to schedule parallel programs.

6.1 Popular Gang Scheduling Algorithms

Among the gang scheduling algorithms, AFCFS and LGFS are the most popular algorithms and we present them next.

6.1.1 AFCFS

The scheduling algorithm AFCFS places the gangs in the run queue in order of their arrival. The scheduling starts from the head of the queue and the gang which can fit into the available cores are scheduled for execution. Unlike FCFS which stops scheduling when it finds the gang which cannot fit into the free cores, AFCFS iterates over the whole run queue and schedules all gangs which can fit into the free cores. The AFCFS algorithm is given in Table. 6.1.

Data Structures: RQ: Queue of gangs

1. **while** ($RQ \neq$ empty) **do**
2. **for** $i = 1$ to size of RQ **do**
3. **if** $RQ[i]$ fits in free cores then schedule $RQ[i]$
4. **end for**
5. **end while**

Table 6.1: AFCFS Gang Scheduling Algorithm

6.1.2 LGFS

Largest gang first scheduling algorithm orders the run queue based on the size of the gangs. The size of the gang is determined by the number of threads. LGFS schedules the gang from the head of the queue(largest gang) until the gang which can fit into the available free cores. LGFS favors the large sized gang over the small sized gang. This will influence the response time of small sized gangs, and that makes the small gangs to wait for longer time to get their turn. Also, when a large size gang arrives, it may overtake these small gangs. The algorithm of LGFS is given in Table. 6.2.

Data Structures: RQ: Queue of gangs

1. **while** ($RQ \neq$ empty) **do**
2. **sort** RQ based on largest gang first
3. **for** $i = 1$ to size of RQ **do**
4. **if** $RQ[i]$ fits in free cores then schedule $RQ[i]$
5. **end for**
6. **end while**

Table 6.2: LGFS Gang Scheduling Algorithm

Parameter	**Value(s)**
Number of cores	200
Time period	1 minute
Tasks per gang	Uniformly distributed over [2..200]
Mean Arrival rate	1.5, 2, 2.5
Arrival distribution	Poisson
Execution rate	2
Execution time distribution	Exponential

Table 6.3: Simulation Parameters

From the literature [53], we note that AFCFS performs better than LGFS in response time in case of lighter workloads with small gangs. Our experiment using the proposed simulation confirms the result.

6.1.3 Simulation Experiments

The parameters used in our simulation are listed in Table. 6.3. We conducted simulation experiments to compute average response time, standard deviation of response time, and average core utilization of AFCFS and LGFS algorithms. The observations are presented next.

- *Observation on Average Response time:* The average response time graphs of AFCFS and LGFS algorithms are shown in Fig. 6.1. The average response time of AFCFS outperforms LGFS for small sized gangs. Since LGFS favors

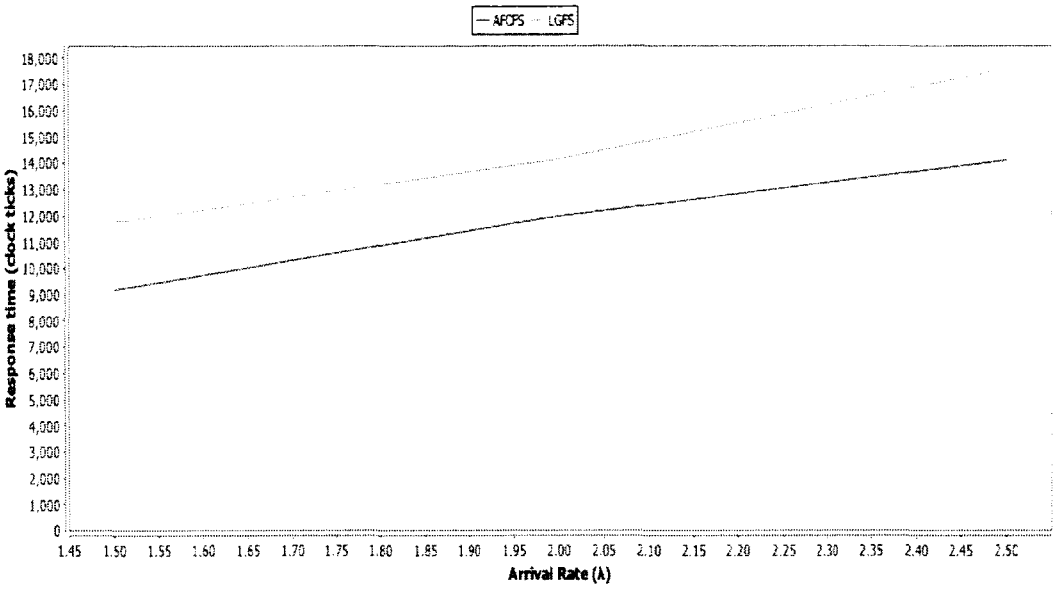

Figure 6.1: Average Response Time for AFCFS and LGFS Algorithms

the large size gangs, it pushes the small sized gangs for longer wait times which is reflected in their average response time. This result confirms the observation from the literature.

- *Observation on Standard deviation in Response time:* The standard deviation response time graphs of AFCFS and LGFS algorithms are shown in Fig. 6.2. The predictability of AFCFS is worse than LGFS. Since the AFCFS favors the small size gang, the large size gangs have to wait longer for their turn for execution which in turn increase the deviation in response time.

- *Observation on Average Core Utilization:* The average core utilization graphs of AFCFS and LGFS algorithms are shown in Fig. 6.3. LGFS performs better than AFCFS, because it favors large jobs which fits into more number of cores and makes the core busy executing these larger jobs. As AFCFS favors smaller jobs, the possibility of core to stay idle is higher.

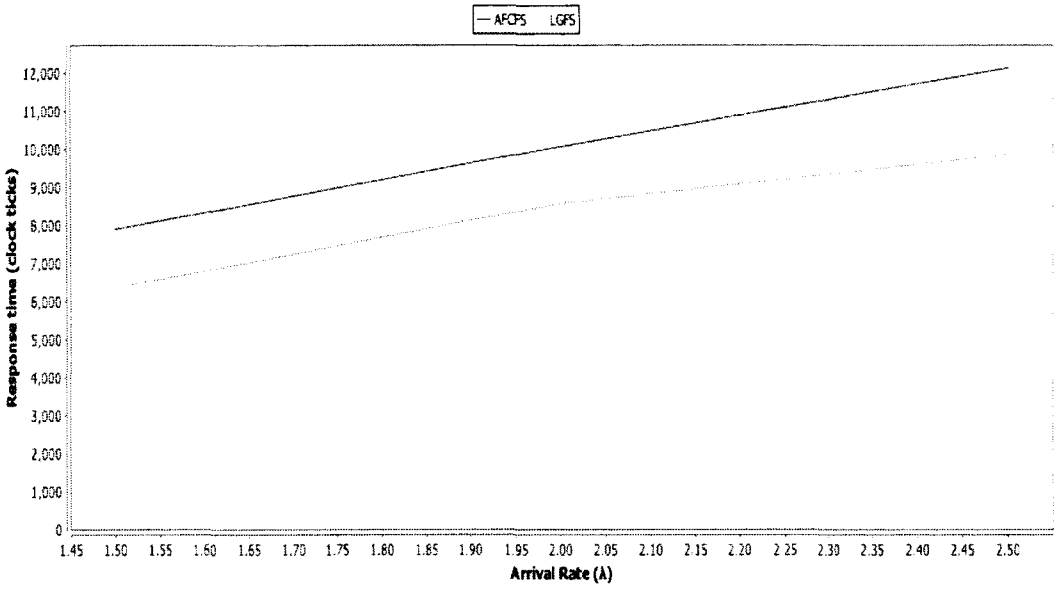

Figure 6.2: Standard deviation of Response Time for AFCFS and LGFS Algorithms

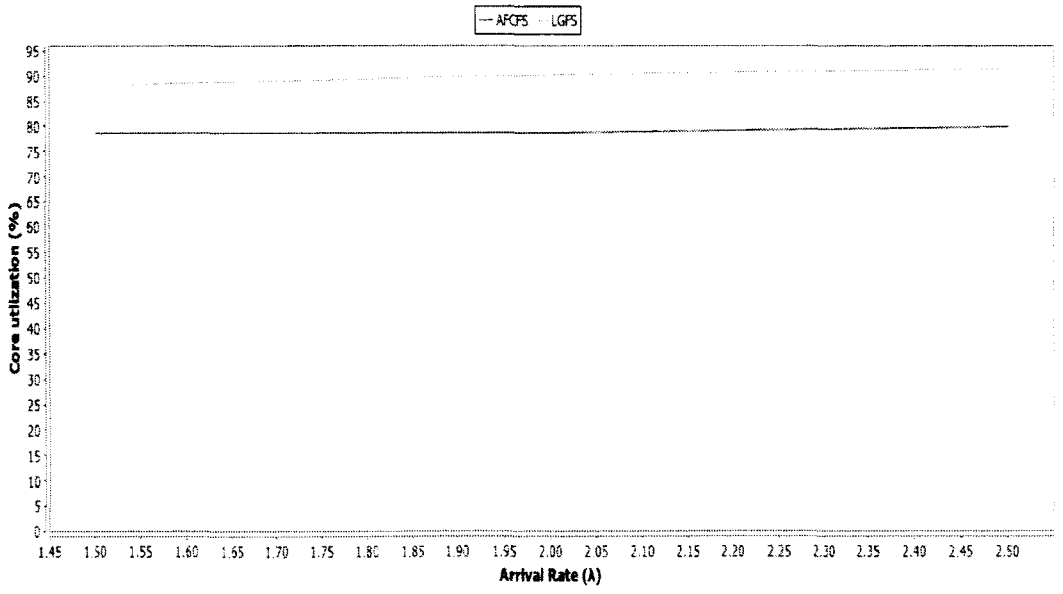

Figure 6.3: Core Utilization of AFCFS and LGFS Algorithms

From the observations shown in Fig. 6.2 and Fig. 6.3, we can see that LGFS is a preferred algorithm, despite AFCFS' low average response time. This is because LGFS utilizes the system resources better and offers better predictability in response time. This proves our earlier point that the average response time is not a desirable metric.

Fairness and predictability are particularly important that the expectation of users under light load is normally high, and failure to provide such guarantee even under light load could expose the system very badly. Therefore, it would be nice to have an algorithm which yields low average response time and standard deviation with high processor utilization. This is the motivation for our algorithm presented next.

6.2 A New Gang Scheduling Algorithm

The gang scheduling algorithms AFCFS and LGFS are susceptible to starvation. To avoid starvation, these algorithms adopt a process migration policy. Process migration may not be even possible between two heterogeneous multicore systems, and is generally expensive even between two homogeneous systems [39,56]. Also, although it alleviates, process migration does not eliminate starvation.

These observations bring us a question: Can we design a gang scheduling algorithm with the following characteristics?

1. Freedom from starvation.
2. Predictable and acceptable response time.
3. Better processor utilization.
4. Simple.

Since AFCFS favors small gangs, the larger gangs are susceptible to starvation or to longer wait times. This is unacceptable particularly in cloud environment where customer satisfaction hugely depends on fairness and predictable response time. In practice, the customers who receive a little faster service (at the expense of others' long wait) may not be overly satisfied [61]. But, the customers who experience unpredictably long delay, on the other hand, will readily notice the unfairness and unpredictable response and that could potentially drive the cloud business in a negative direction. Therefore, in addition to fast response and high processor utilization, minimal variance in response is extremely important for better cloud services.

6.3 The Algorithm

The gang scheduling algorithm proposed in this thesis combines the ideas of AFCFS and priority boosting. In the AFCFS algorithm proposed for multicore clusters in [53], each multicore has a run queue and all gangs stay in the run queue until it gets a chance to execute. The scheduler always chooses the next fit gang from the run queue so that overall response time is reduced. Such behavior degrades the overall core utilization which in turn increases the variation in response time as seen in the experiments.

The proposed algorithm uses an additional variable for each gang which stores the information about how many gangs bypassed it for execution when it stayed in run queue. We call that variable *'Bypass count'*. When the gang's bypass count reaches the threshold value T, it gets the highest priority to schedule next. This pushes other gangs to force wait until the highest priority gang gets scheduled. The proposed algorithm is given in Table. 6.4.

A new gang joins RQ, and its bypass count is set to zero. Whenever a gang

is scheduled, the bypass count of gangs precedes the scheduled gang in RQ will be incremented by 1. At any time, gang in RQ with bypass count greater or equal to the threshold value has the highest priority over other gangs. This guarantees that the gangs will be served in a predictable time period. When there is no gang with bypass count greater or equal to the threshold value, it acts as AFCFS algorithm.

When enough cores are not available to schedule the highest priority gang, the system has to wait for some of the currently executing gangs to leave, and this delay is unavoidable to assure fairness and predictable response. The simulation results show that such a wait rarely happens.

Data Structures: RQ: Queue of gangs;T: Threshold value; $i.bpc$: bypass count of gang i

```
1. while (RQ ≠ empty ) do
2.    for i = 1 to size of RQ do
3.       if RQ[i].bpc ≥ T then wait until RQ[i] fits in free cores
4.       if RQ[i] fits in free cores then
5.          for k = 1 to i − 1 do RQ[k].bpc + + end for
6.          schedule RQ[i]
7.       end if
8.    end for
9. end while
```

Table 6.4: New Gang Scheduling Algorithm

Note: The proposed algorithm becomes AFCFS if the threshold is set to ∞. When the threshold is 0, it emulates FCFS algorithm. So, choosing a proper threshold is the key of the proposed algorithm.

6.4 Simulation Experiments

As explained earlier, the proposed algorithm tries to achieve predictable and fast response for gangs (users) and better utilization for the system.

Parameter	Value(s)
Number of cores	200
Time period	1 minute
Tasks per gang	Uniformly distributed over [2..200]
Mean Arrival rate	5, 7.5, 10
Arrival rate distribution	Poisson
Execution rate	2
Execution rate distribution	Exponential
Threshold	700

Table 6.5: Simulation Parameters

6.4.1 Simulation Setup

We used the simulation parameters listed in Table 6.5 for our experiments. To keep the results generic, the execution is shown in terms of simulation clock ticks. Through simulation study, we computed average response time, standard deviation in response time, average core utilization, and bypass count for the gangs. The observations are presented next.

- *Observation on Average Response Time:* The average response time graphs of AFCFS and the proposed algorithms are shown in Fig. 6.4. The average response time of the proposed algorithm is better than that of AFCFS. This is because, whenever the gangs' bypass count reaches the threshold, it guarantees the gang to schedule next which reduces the response time of long waiting gangs. Choosing a proper threshold value is crucial. Choosing a small number will unnecessarily make others gangs to wait more often, which will in turn increase the average response time. For our experiments, we have chosen 700[1] bypass count as the threshold value. From the Fig. 6.4, it is clear that the average response time of proposed algorithm performs better than AFCFS consistently.

[1]The threshold value is derived from the repetitive experiments for consistent behavior.

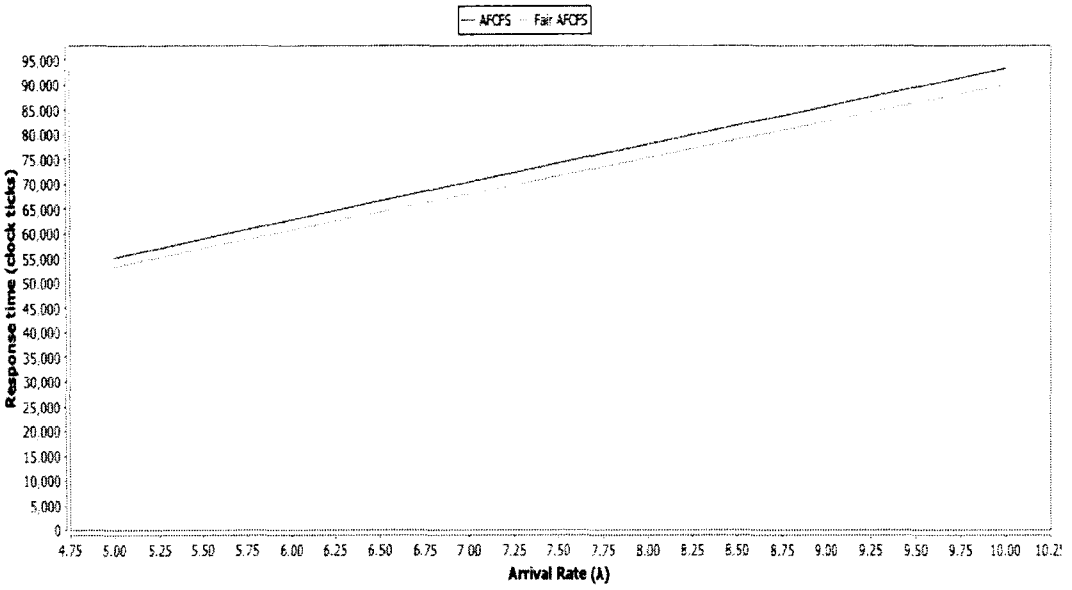

Figure 6.4: Average Response Time of AFCFS and Proposed Algorithms

- *Observation on Standard deviation of Response time:* The average turnaround time graphs of AFCFS and the proposed algorithms are shown in Fig. 6.5. The proposed algorithm offers better predictability in response time than AFCFS algorithm. Since, the proposed algorithm avoids the longer wait times, the predictability in response time will be lower than AFCFS. By controlling the bypass threshold value, better predictability may be assured.

- *Observation on Average Core Utilization:* The average core utilization graphs of AFCFS and the proposed algorithms are shown in Fig. 6.5. The proposed algorithm outperforms AFCFS algorithm. This is because, AFCFS favors only small gangs but the proposed algorithm favors all sized gangs once the threshold is reached.

- *Observation on Bypass count graph:* The bypass count graphs of AFCFS and the proposed algorithm are shown in Fig. 6.7. The proposed algorithm shows

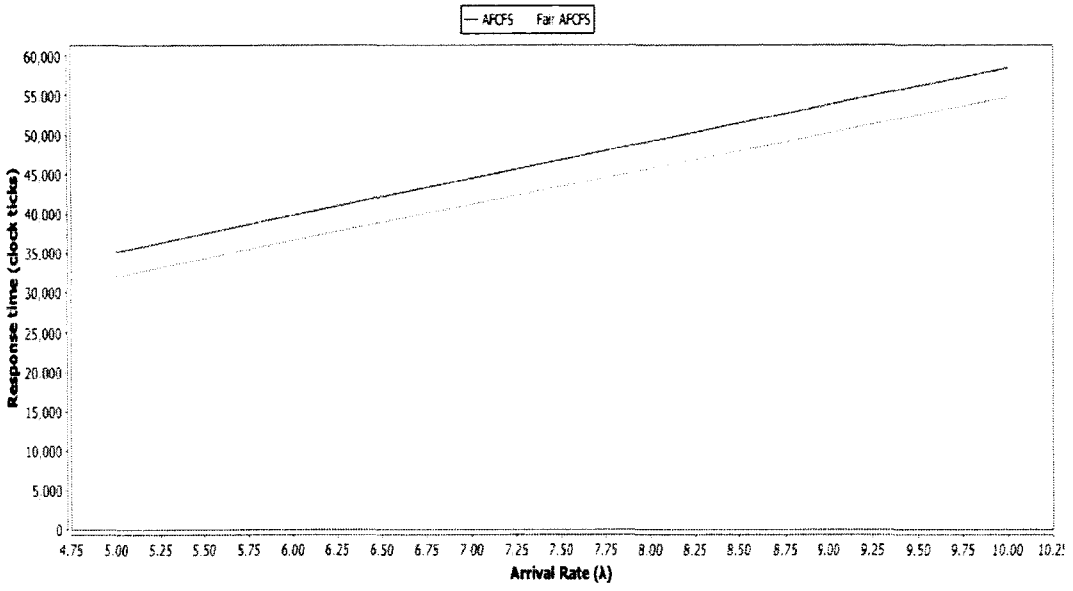

Figure 6.5: Standard deviation of Response Time of AFCFS and Proposed Algorithms

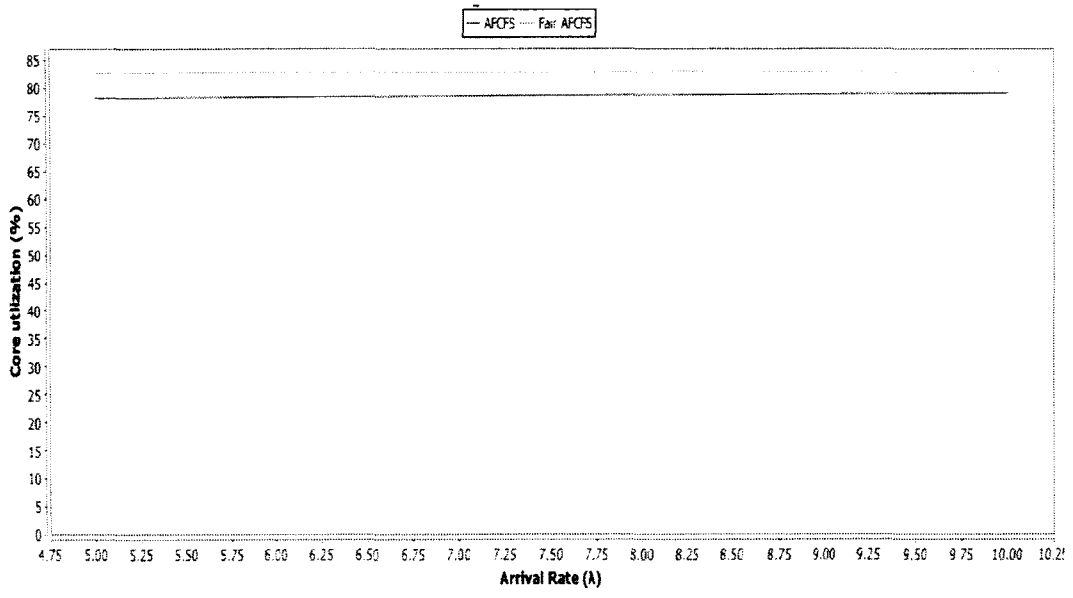

Figure 6.6: Average Core Utilization of AFCFS and Proposed Algorithms

the fairness among gangs, once it reaches the threshold, it starts giving the priority for long waited gangs. The longer wait time is completely avoided in the proposed algorithm, which makes our algorithm interesting.

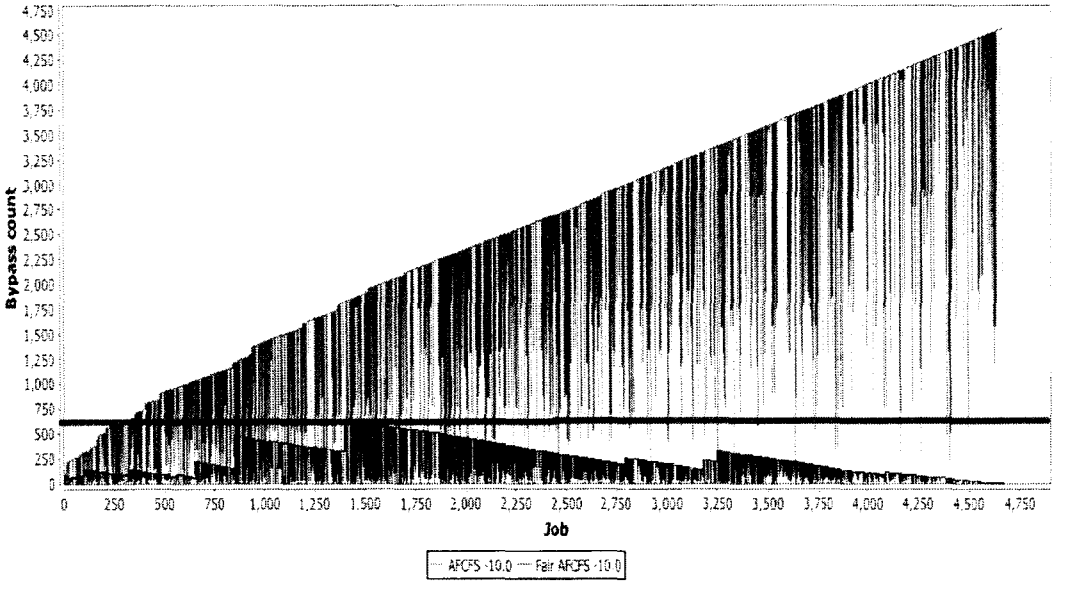

Figure 6.7: Bypass Count of Gangs of AFCFS and Proposed Algorithms

From these observations, we conclude that, the proposed algorithm outperforms AFCFS in all three metrics, and of course solves the starvation problem completely.

6.5 Summary

In this chapter, we proposed a fair and efficient gang scheduling algorithm for multicore processors. The algorithm is simple, fair, and gives predictable performance. Such a predictable performance is attractive from the service point of view. Since this algorithm solves the starvation problem locally without using process migration, it is highly scalable and attractive for cloud computing involving a large number of multicore processors.

Chapter 7

Conclusion and Future Directions

Recently, multicore processors and software development for multicore systems have received increasing attention from the research community. The contributions of this thesis are: (i) the design and implementation of a flexible multicore scheduler simulation framework; (ii) illustration of the power and flexibility of the proposed framework by simulating the scheduling algorithms of Linux and Solaris, and a simulation study of two gang scheduling algorithms; and (iii) a new gang scheduling algorithm and its performance study. The experience gained by developing this simulator is very rich. It involved software design, implementation, algorithm discovery and design, and performance analysis.

The proposed simulator can be used for rapid simulation studies of multicore scheduling algorithms, and that can provide initial insights on how the proposed algorithm will perform in practical multicore systems. These insights will be helpful, not only in testing the performance of the proposed algorithm, but also to identify the bottlenecks and offer guidelines for improvements. Also, from our experience, we believe that the simulator can be used to develop new scheduling algorithms by

experimenting and identifying the limitations of the existing algorithms.

Although, initially we did not expect to propose a new multicore scheduling algorithm, we finally ended up designing one for an important class of parallel applications called gangs. We compared the performance of the new gang scheduling algorithm with the known best algorithm in its category. The proposed algorithm seems to perform better.

7.1 Future Directions

We believe the proposed scheduling framework for multicore systems is an important first step in the performance study of multicore scheduling algorithms. There are many directions in which the work presented in this thesis can be expanded to study the performance of scheduling algorithms deeper and more accurately. We outline some of them next.

- Modeling workload could be refined and improved.
- Modeling I/O waits could be refined and improved.
- The modeling of cache can be refined and improved.
- The framework can be expanded to model heterogeneous cores.
- Cache effect can be inferred from the values of hardware performance counters available in the recent multicore machines, and used in scheduling simulations.
- More statistical metrics can be included.
- In cloud computing context, modeling clusters with many chips and its associated cores would be more interesting.

- The proposed gang scheduling algorithm works better for light loads. Is there a gang scheduling algorithm that can perform better under all workload conditions? This is an interesting research question to be explored.

I would like to continue to work on some of these directions in the future.

Appendix

Tables used by Solaris Scheduler

Solaris scheduler uses the information given in the following four tables -

1. **Priority Range Table** (Table 7.1) - specifies the priority range for the scheduling classes.
2. **Dispatch Table for Real-time Tasks** (Table 7.2) - defines the time quanta of real time scheduling class.
3. **Dispatch Table for Fixed Priority Tasks** (Table 7.3) - defines the time quanta of fixed priority scheduling class.
4. **Dispatch Table for Normal Tasks** (Table 7.4) - defines the time quanta of time sharing and interactive scheduling classes.

Scheduling class	Global priority range	User level priority range
Realtime	100 - 159	-
System	60 - 99	-
Fair share	0 - 59	0 - 59
Fixed priority	0 - 59	0 - 60
Time share	0 - 59	-60 - 60
Interactive	0 - 59	-60 - 60

Table 7.1: Solaris 10 Scheduling Classes Priority Range

Quanta	Priority
100	100
80	110
60	120
40	130
20	140
10	150
10	159

Table 7.2: Dispatch Table for RT Scheduling Class

Quanta	Priority
0	20
10	16
20	12
30	8
40	4
59	2

Table 7.3: Dispatch Table for FX Scheduling Class

Quanta	Priority on Quanta Expiry	Priority on IO Return	Wait Threshold	Priority on Wait	Priority
20	0	50	0	50	0
16	0	51	0	51	10
12	10	52	0	52	20
8	20	53	0	53	30
4	30	55	0	55	40
2	49	59	3200	59	59

Table 7.4: Dispatch Table for TS and IA Scheduling Classes

Tick Processing and Update Processing

Here we present two key routines used in Solaris scheduling.

Tick Processing

Tick processing will be executed for every scheduling tick. This method is mainly responsible for managing the time quanta and preemption control. The overview of what tick processing is doing is given in Table. 7.5. When the tick processing method is invoked, it first checks whether the executing thread is in system mode. This is because, system threads are scheduled for its full execution time and preemption of system threads are not allowed. Other than system threads, all other scheduling classes are preemption enabled. The main task of the tick processing is to manage the allocated time quanta. First, it decrements the quanta allocated for the thread and checks whether the thread executed for it's whole time quanta. If so, it additionally checks whether the preemption control is enabled for the thread. The preemption control is a variable to give few more time for a thread when it finishes it allotted time quanta if preemption control is enabled. This variable can be configured. If the preemption control is not enabled for the thread, then the first task is to re-assign the priority of thread from their dispatch table in case of TS and IA class thread.

RT and FX class threads are maintained with the same priority. Then, it invokes the preemption and places the thread in their dispatch queue based on their priority. When the time quanta is not over, then it checks whether the thread is going for I/O and places the thread in the sleep queue. The method also checks for any highest priority thread in the dispatch queues and preempts the current thread. When the thread is preempted by a high priority thread, then the priority of the current thread won't change.

Data Structures: Thread t

1. **if** thread t is not in system priority **then**
2. decrement the time quanta $t.quanta$
3. **if** $(t.quanta \leq 0)$ **then**
4. **check** thread preemption control enabled **then**
5. give additional time quanta for execution
6. **else**
7. **if** $t \in TS$ or IA scheduling class **then**
8. re assign $t.priority$ from dispatch_table_ts with the value of ts_tqexp
9. **end if**
10. enable preemption and place thread t in their dispatch queues
11. **end if**
12. **if** t going for I/O **then**
13. enable preemption and place thread t in sleep queue
14. **end if**
15. **if** $t.priority <$ highest thread's priority in dispatch queues **then**
16. enable preemption and place the thread t in dispatch queue
17. **end if**
18. **end if**

Table 7.5: Tick Processing of Solaris Scheduling Algorithm

Update Processing

Update processing method is invoked only for time sharing and interactive scheduling classes. The main purpose of this method is to boost up the priority. The higher level implementation of update processing is given in Table. 7.6.

Update processing method will be invoked periodically. This method will iterate over the dispatch and sleep queues to increment the waiting time of the threads. Once the waiting time of a thread reaches maximum wait time specified in dispatch table, the thread's priority will be boosted up by the ts_lwait value. This is done to provide fairness among the threads.

Data Structures: Dispatch queues - DQ_TS, DQ_IA, sleep queue - SQ, Thread t

1. **for each** thread t is not in $DQ\_TS \wedge DQ\_IA \wedge SQ$ **do**
2. increment $t.wait$ value by 1
3. **if** ($t.wait \geq$ max_wait) **then**
4. boost up thread priority from dispatch_table_ts with the value of ts_lwait
5. **end if**
6. **end for**

Table 7.6: Update Processing of Solaris Scheduling Algorithm

Bibliography

[1] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *SIGOPS Operating System Review*, vol. 43, pp. 76–85, April 2009.

[2] A. Fedorova, M. Seltzer, and M. D. Smith, "Cache-fair thread scheduling for multicore processors," tech. rep., Harvard University, 2006.

[3] B. Wilkinson and M. Allen, *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers Second Edition.* Pearson Prentice Hall, 2005.

[4] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science Engineering, IEEE*, vol. 5, pp. 46 –55, jan-mar 1998.

[5] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: The Complete Reference.* Cambridge, MA, USA: MIT Press, 1995.

[6] S. Peter, A. Schupbach, P. Barham, A. Baumann, R. Isaacs, T. Harris, and T. Roscoe, "Design principles for end-to-end multicore schedulers," in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[7] C. Kesselman and I. Foster, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Nov. 1998.

[8] D. Wentzlaff, C. G. III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. E. Miller, and A. Agarwal, "An operating system for multicore and clouds: Mechanisms and implementation," in *SoCC*, pp. 3–14, 2010.

[9] "The future accelerated: Multi-core goes mainstream, computing pushed to extremes." Intel Newsroom, September 2011.

[10] J. C. Mogul, A. Baumann, T. Roscoe, and L. Soares, "Mind the gap: reconnecting architecture and os research," in *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2011.

[11] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, (New York, NY, USA), pp. 29–44, ACM, 2009.

[12] S. Zhuravlev, J. C. Saez, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Computing Surveys*, In Press.

[13] D. Wentzlaff, C. G. III, N. Beckmann, A. Belay, H. Kasture, K. Modzelewski, L. Youseff, J. E. Miller, and A. Agarwal, "Fleets: Scalable services in a factored operating system," tech. rep., CSAIL Massachusetts Institute of Technology, 2011.

[14] A. Schbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs, "Embracing diversity in the barrelfish manycore operating system," in *In Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[15] A. Belay, D. Wentzlaff, and A. Agarwal, "Vote the os off your core," tech. rep., CSAIL Massachusetts Institute of Technology, 2011.

[16] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs, "Your computer is already a distributed system. why isn't your os?," in *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, (Berkeley, CA, USA), pp. 12–12, USENIX Association, 2009.

[17] S. Panneerselvam and M. M. Swift, "Dynamic processors demand dynamic operating systems," in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, (Berkeley, CA, USA), pp. 9–9, USENIX Association, 2010.

[18] I. Kuz, Z. Anderson, P. Shinde, and T. Roscoe, "Multicore os benchmarks: we can do better," in *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2011.

[19] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: an operating system for many cores," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, (Berkeley, CA, USA), pp. 43–57, USENIX Association, 2008.

[20] S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek, "Reinventing scheduling for multicore systems," in *Proceedings of the 12th conference on Hot topics in oper-*

ating systems, HotOS'09, (Berkeley, CA, USA), pp. 21–21, USENIX Association, 2009.

[21] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz, "Tessellation: space-time partitioning in a manycore client os," in *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2009.

[22] D. Wentzlaff, C. G. III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal, "A unified operating system for clouds and manycore: fos," *1st Workshop on Computer Architecture and Operating System co-design (CAOS)*, 2010.

[23] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Transactions on Computer Systems*, vol. 28, pp. 8:1–8:45, December 2010.

[24] A. Fedorova, C. Small, D. Nussbaum, and M. Seltzer, "Chip multithreading systems need a new operating system scheduler," in *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, EW 11, (New York, NY, USA), ACM, 2004.

[25] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Akula: a toolset for experimenting and developing thread placement algorithms on multicore systems," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 249–260. ACM, 2010.

[26] J. M. Calandrino, D. P. Baumberger, T. Li, J. C. Young, and S. Hahn, "Linsched: The linux scheduler simulator," in *ISCA PDCCS* (J. Jacob and D. N. Serpanos, eds.), pp. 171–176, ISCA, 2008.

[27] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation: the simos approach," *Parallel Distributed Technology: Systems Applications, IEEE*, vol. 3, pp. 34 –43, winter 1995.

[28] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, pp. 50 –58, feb 2002.

[29] 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*, PACT '08, (New York, NY, USA), pp. 220–229, ACM, 2008.

[30] T. C. Xu, P. Liljeberg, and H. Tenhunen, "Process scheduling for future multicore processors," in *Proceedings of the Fifth International Workshop on Interconnection Network Architecture: On-Chip, Multi-Chip*, INA-OCMC '11, (New York, NY, USA), pp. 15–18, ACM, 2011.

[31] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov, "A comprehensive scheduler for asymmetric multicore systems," in *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, (New York, NY, USA), pp. 139–152, ACM, 2010.

[32] V. Kazempour, A. Kamali, and A. Fedorova, "Aash: an asymmetry-aware scheduler for hypervisors," in *Proceedings of the 6th ACM SIGPLAN/SIGOPS inter-*

national conference on Virtual execution environments, VEE '10, (New York, NY, USA), pp. 85–96, ACM, 2010.

[33] S. Hofmeyr, C. Iancu, and F. Blagojević, "Load balancing on speed," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, (New York, NY, USA), pp. 147–158, ACM, 2010.

[34] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry, "Decoupling contention management from scheduling," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, (New York, NY, USA), pp. 117–128, ACM, 2010.

[35] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, (New York, NY, USA), pp. 245–254, ACM, 2009.

[36] L. Tang, J. Mars, and M. L. Soffa, "Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures," in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (co-located with PLDI 2011)*, (New York, NY, USA), pp. 12–21, ACM, 2011.

[37] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts,*. John Wiley & Sons, Inc., 2010.

[38] S. Haldar and A. Aravind, *Operating Systems*. Pearson Education, 2010.

[39] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Computing Surveys*, vol. 32, pp. 241–299, Sept. 2000.

[40] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *Proceedings of the IEEE Distributed Computing Systems*, pp. 22 – 30, 1982.

[41] L. Chai, Q. Gao, and D. K. Panda, "Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system," in *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '07, (Washington, DC, USA), pp. 471–478, IEEE Computer Society, 2007.

[42] I. A. Moschakis and H. D. Karatza, "Evaluation of gang scheduling performance and cost in a cloud computing system," *Journal of Supercomputing*, vol. 59, pp. 975–992, Feb. 2012.

[43] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," tech. rep., 1997.

[44] M. Moudgill, P. Bose, and J. Moreno, "Validation of turandot, a fast processor model for microarchitecture exploration," in *Performance, Computing and Communications Conference, 1999 IEEE International*, pp. 451 –457, feb 1999.

[45] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "Cotson: infrastructure for full system simulation," *SIGOPS Operating System Review*, vol. 43, pp. 52–61, Jan. 2009.

[46] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the simos machine simulator to study complex computer systems," *ACM Trans. Model. Comput. Simul.*, vol. 7, pp. 78–103, Jan. 1997.

[47] *AMD Developer Central. AMD SimNow Simulator, http://developer.amd.com/tools/simnow/pages/default.aspx.*

[48] A. Batat and D. G. Feitelson, "Gang scheduling with memory considerations," in *in Proc. of the 14th Intl. Parallel and Distributed Processing Symp., 2000*, pp. 109–114, 2000.

[49] K. Hyoudou, Y. Kozakai, and Y. Nakayama, "An implementation of a concurrent gang scheduler for a pc-based cluster system," *Systems and Computers in Japan*, vol. 38, pp. 39–48, Mar. 2007.

[50] H. D. Karatza, "Scheuling gangs in a distributed system," *International Journal of Simulation*, vol. 7(1), pp. 15–22, 2006.

[51] Z. C. Papazachos and H. D. Karatza, "The impact of task service time variability on gang scheduling performance in a two-cluster system," *Simulation Modelling Practice and Theory*, vol. 17, no. 7, pp. 1276 – 1289, 2009.

[52] Z. C. Papazachos and H. D. Karatza, "Gang scheduling in a two-cluster system implementing migrations and periodic feedback," *SIMULATION*, 2010.

[53] Z. C. Papazachos and H. D. Karatza, "Gang scheduling in multi-core clusters implementing migrations," *Future Generation Computer Systems*, vol. 27, no. 8, pp. 1153 – 1165, 2011.

[54] Y. Wiseman and D. G. Feitelson, "Paired gang scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, pp. 581–592, June 2003.

[55] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam, "An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration," in *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, JSSPP '01, (London, UK), pp. 133–158, Springer-Verlag, 2001.

[56] S. Frechette and D. R. Avresky, "Method for task migration in grid environments," in *Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications*, NCA '05, (Washington, DC, USA), pp. 49–58, IEEE Computer Society, 2005.

[57] R. McDougall and J. M. and, *Solaris Internals (2nd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006.

[58] C. S. Pabla, *Completely Fair Scheduler, http://www.linuxjournal.com/magazine/completely-fair-scheduler*, 2009.

[59] I. Stojmenovic, "Simulations in wireless sensor and ad hoc networks: matching and advancing models, metrics, and solutions," *Communications Magazine, IEEE*, vol. 46, pp. 102 –107, december 2008.

[60] K. Sankaralingam and R. H. Arpaci-Dusseau, "Get the parallelism out of my clouds," in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, (Berkeley, CA, USA), pp. 8–8, USENIX Association, 2010.

[61] G. Ghinea and S. Chen, "Perceived quality of multimedia educational content: A cognitive style approach," *Multimedia Systems*, vol. 11, pp. 271–279, 2006. 10.1007/s00530-005-0007-8.

[62] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*. John Wiley & Sons, Inc., 2000.

[63] *Completely Fair Scheduling (CFS) Class, http://lxr.linux.no/linux+v3.3.1/kernel/sched/fair.c*.