UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

PREDICTION MODELS FOR ESTIMATING THE EFFICIENCY OF DISTRIBUTED MULTI-CORE SYSTEMS

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

By

KHONDKER SHAJADUL HASAN
Norman, Oklahoma
2014

PREDICTION MODELS FOR ESTIMATING THE EFFICIENCY OF DISTRIBUTED MULTI-CORE SYSTEMS

A DISSERTATION APPROVED FOR THE SCHOOL OF COMPUTER SCIENCE

BY

Dedication

This dissertation is dedicated to

My parents
Khundker Manzur Murshed and Razia Murshed,
my wife Afroza Shajadul,
my son Shezad, daughters Samreena and Ariana,
and my brothers and sister

For
Encouragement, belief, and patience

Acknowledgements

This work has been possible because of a number of individuals. First of all, I would like to express my sincere gratitude to my dissertation committee chair Dr. John K. Antonio and co-chair Dr. Sridhar Radhakrishnan for their expert advice, patience, and supervision throughout my Ph.D. studies, which made this work possible.

I am truly grateful to my committee members: Dr. Mohammed Atiquzzaman, Dr. Ronald D. Barnes, and Dr. Dean Hougen for their valuable advice and time to review this dissertation. I would like to specially thank Dr. Dean Hougen for investing considerable amount of time for providing detailed comments and suggestions that was very helpful to improve the quality of this dissertation.

I am also thankful to all the faculty and staff members of the School of Computer Science at the University of Oklahoma for providing a supportive environment for my study. Moreover, numerous discussions with my fellow researchers Amlan Chatterjee, Nicolas G. Grounds, Asif M. Adnan, Dr. Mahendran Veeramani, Jonathan Mullen, and Chandrika Satyavolu and their technical expertise and valuable support has led to the improvements of this work.

I would like to acknowledge the financial support of MSCI Inc, Norman to carry out the initial part of this research work.

I would also like to thank my wife Afroza Shajadul and my childrens Shezad, Samreena, and Ariana for accompanying and supporting me throughout all the tough times. Last but not the least, I would like to extend my gratitude towards my parents, sister, and brothers specially Sherajul for providing moral support to me and my family.

Contents

List of Tables

List of Figures

Abstract

The efficiency of a multi-core architecture is directly related to the mechanisms that map the threads (processes in execution) to the cores. Determining the resource availability (CPU and main memory) of the multi-core architecture based on the characteristics of the threads that are in execution is the art of system performance prediction. In this dissertation we develop several prediction models for multi-core architectures and perform empirical evaluations to demonstrate the accuracy of these models.

Prediction of resource availability is important in the context of making process assignment, load balancing, and scheduling decisions. In distributed infrastructure, resources are allocated on demand on a chosen set of compute nodes. The nodes chosen to perform the computations dictate the efficiency by which the jobs assigned to them will be executed. The prediction models allows us to estimate the resource availability without explicitly querying the individual nodes. With the model in hand and knowledge of the jobs (such as peak memory requirement and CPU execution profile), we can determine the appropriate compute nodes for each of the jobs in such a way that it will improve resource utilization and speed job execution.

More specifically, we have accomplished the following as part of this dissertation:

(a) Develop mathematical models to estimate the upper- and lower-limits of CPU and memory availability for single- and multi-core architectures.

(b) Perform empirical evaluation in a heterogeneous environment to validate the accuracy of the models.

(c) Introduce two task assignment policies that are capable of dispatching tasks to distributed compute nodes intelligently by utilizing composite prediction and CPU usage models.

(d) Propose a technique and introduce models to identify combinations of parameters for efficient usage of GPU devices to obtain optimal performance.

Chapter 1

Introduction

The success of approaches for assigning threads (or processes) to multi-core and many-core systems relies on the existence of reasonably accurate models for estimating resource availability. This is because there is a strong relationship between a thread's total execution time and the availability of CPU and memory resources used for its execution [1]. Therefore, predicting the resource availability that results when threads are assigned to a processor is a basic problem that arises in many important contexts. Accurate resource availability prediction model is important because there exists a wide range of applications and scientific models (e.g., geological, meteorological, economical and others) that requires extensive use of both CPU and memory resources, repeatedly. These models fall into this category, where the program remains the same and the data on which it operates changes over time. The distributed task assignment problem in general is NP-Hard and it is further complicated by the changing dynamics (changes to CPU and main memory availability) of the compute nodes [1].

1.1 Overview

In time-shared systems, the utilization of the CPU and the speed of the computer's response to its users have improved as a result of CPU scheduling [2]. To realize this increase in performance, it is often customary to keep several processes running in the system [3]. For a process to execute, it is necessary for the system to swap in the required portion of code and data of the process in the physical memory (i.e., necessary pages containing executable codes and data are placed into primary memory). Additional pages can be loaded when they are demanded during program execution.

As the degree of multi-programming has increased significantly, over-allocating memory may result in severe performance degradation. Demand paging saves the I/O necessary to load pages that are never used but increases the probability of memory overrun. This situation requires added attention and is considered as a matter of concern in this dissertation because the memory overrun triggers page swapping from primary memory to backing store (victim pages are selected from existing processes) which can escalate the process execution time immensely. If one access out of $1,000$ causes a page fault, the system can be slowed down by a factor as high as 40 because of the page fault service time. Thus, the effective access time is directly proportional to the page-fault rate [13, 14].

In this dissertation, an approach for predicting the efficiency of multi-core processing associated with a set of concurrent tasks (or processes) with varied CPU requirements is introduced. Prediction of *CPU availability* is important in the context of making process assignment, load balancing, and scheduling decisions in distributed systems. Theoretically derived upper- and lower-bound formulas for estimating CPU availability are introduced. Input parameters to the formulas include: number of cores; number of threads; core hyper-threading; and the unloaded CPU usage factor for each thread. From the empirical results, a model for *average CPU availability* is introduced for the set of applications that require a single predicted value (instead of bounds).

In addition to CPU availability, *memory performance model* for estimating the execution efficiency of tasks due to primary memory availability is introduced. A memory performance model is derived to predict the execution efficiency of concurrent tasks when the available memory is less than required amount of memory. Parameters used in this model are: required memory by processes; available memory frames; memory access time; and backing store overhead. The CPU and Memory availability models are then used as building blocks to derive the *Composite resource availability* prediction model. Through empirical studies, it is shown that the proposed bounds processing efficiency are consistently tight. The proposed composite prediction model provides a basis of an empirical model for estimating execution efficiency of processes while CPU and memory resources are uncertain.

By utilizing the introduced composite prediction model, two policies for efficiently schedule *heterogeneous tasks* in a distributed environment are introduced and extensively evaluated empirically. Given a set of tasks each with varied CPU and main memory requirements, and a cluster of compute nodes (which are significantly less than the number of tasks), our goal is to find an assignment of tasks to compute nodes such that the total time taken to execute all the tasks is minimized. The key challenge is to determine the CPU availability prior to placement of tasks into the run queue. A new model is presented in this dissertation to predict the *CPU availability for a new task* (instead of a group of tasks) before placing the task into the run-queue. For all empirical evaluations on UNIX systems, benchmark programs are deployed that are both synthetic (allowing us to control the CPU and memory requirements) and real-worlds tasks including prime number generator, merge sort, image rendering, and others for validating introduced resource availability and task assignment models. The empirical results are then compared with Terascale Open-source Resource and QUEue Manager (TORQUE) and it's observed that both introduced model outperforms the popular TORQUE scheduler.

Using Graphics Processing Units (GPUs) to solve general purpose problems has received significant attention both in academia and industry. Harnessing the power of these devices however requires knowledge of the underlying architecture and the programming model. Because of the increasing usability of GPUs, an analytical model to predict the performance of GPUs for computationally intensive tasks is also introduced. The proposed model for GPUs provides alternative possible combinations of the parameters for efficient usage of the device and obtaining optimal performance. The study can also be used in the context of heterogeneous environments where multiple GPU devices with different hardware configurations are employed.

1.2 Motivation and Problem Statement

Multi-threading is a common technique used for exploiting performance from multi-core processors. When the number of threads assigned to a multi-core processor is less than or equal to the number of CPU cores associated with the processor, then the performance of the CPU is predictable, and is often nearly ideal. When the number of assigned threads is more than the number of CPU cores, the resulting CPU performance can be more difficult to predict [1]. For example, assigning two CPU-bound threads to a single core results in CPU availability of about 50%, meaning that roughly 50% of the CPU resource is available for executing either thread. Alternatively, if two I/O-bound threads are assigned to a single core, it is possible that the resulting CPU availability is nearly 100%, provided that the usage of the CPU resource by each thread is fortuitously interleaved. However, if the points in time where both I/O-bound threads do require the CPU resource overlap (i.e., they are not interleaved), then it is possible (although perhaps not likely) that the CPU availability of the two I/O bond threads could be as low as 50% [7]. Therefore, considering the number of processes in the run queue itself as a load index is not generally sufficient.

Existing prediction models typically assume CPU resources are equally distributed among all processes in the run queue by following a Round Robin (RR) scheduling technique [1, 13]. These models use the number of processes in the run queue as the system load index. As a result, the CPU availability prediction for a newly arriving process when there are currently N processes in the run queue is simply $1/(N + 1)$. This predictor is only accurate for CPU-bound processes, which share CPU resources in a balanced manner; consistent with the RR model assumption. But, when the processes also require I/O resources, this approach fails to provide accurate predictions. Thus, when there are processes in the run queue that require CPU and I/O resources, a more complex model is necessary to describe (and predict) how the CPU is shared.

For any prediction approach, it can be useful to know *a priori* certain characteristics of tasks that are planned to be assigned to the system. As an example, it is useful to know the maximum amount of main memory a task will consume during its execution (*memory requirement*). This information is useful to forecast the amount of time that may be consumed for memory paging activities. It is also useful to know the *CPU requirement* of the task, which is the fraction of time a task requires the CPU. While a priori information such as memory and CPU requirements of tasks are useful, it is easier to obtain in some cases and more difficult in others. For example, in the case of Merge sort we can approximately determine the memory requirement based on the number of elements to be sorted [17]. For tasks such as generating prime numbers, computing Fast Fourier Transforms, and related others require significant use of the CPU. One can also know the CPU and memory requirements of a task based on the information gathered from its earlier executions. The execution of many scientific models (economic, meteorological, and others) fall into this category, where the program remains the same and the data on which it operates changes over time.

Next to focus on massive parallel programming domain, GPUs have established their usefulness for solving general purpose large problems that requires massive parallel execution. The availability of GPUs as commodity hardware and co-processors to CPUs, and the relative low cost-to-performance ratio has propelled these devices to the forefront or research in both academia and industry. However, in reality, to exploit the full potential of these devices, understanding the underlying architecture and the basics of the programming model are required. Because of this paradigm shift which has left developers striving for better performance from the available hardware has provided us a scope to analyze and develop analytical models to predict the performance of GPUs for computationally intensive tasks.

1.3 Objectives of this Research

The *objectives* of this research are as follows:

- The first objective of this research is to theoretically derive a CPU availability prediction model that can estimate the available CPU for a batch of tasks for single- and multi-core systems. The idea is to investigate the actual CPU usage for a set of benchmark threads and check whether they fall inside the introduced upper- and lower-bounds. Moreover, accuracy of the CPU availability model is performed in Windows systems with Java programming language and Linux systems with C programming language.

- The second objective of this research is to predict the execution efficiency of running threads when the required amount of memory by running threads are more than available primary memory. To achieve this goal, we propose a Memory performance model and perform extensive empirical studies for validating the model.

- The third objective of this research is to combine the effects of two primary resources of compute systems (CPU and Memory) non-trivially and derive a Composite resource prediction model by utilizing the CPU availability and memory models. We verify the accuracy of the composite model by deploying real-world benchmark programs, thereby ensuring the reliability and usefulness for real-world applications.

- The fourth objective of this research is to use the composite prediction model as a building block to introduce distributed task assignment policies. The idea is to intelligently distribute jobs from the queue to distributed multi-core compute nodes so that the overall execution time is reduced. Here, the key challenge is to dynamically estimate available resource of nodes to make correct task assignment decision. To achieve this goal, the new CPU usage prediction model is derived to dynamically estimate resource availability whenever a new task arrives or an old task terminates.

- Finally, using Graphics Processing Units (GPUs) to solve general purpose problems has received significant attention both in academia and industry. Harnessing the power of these devices, however, requires knowledge of the underlying architecture and the programming model. Our fifth objective is to provide guidelines to programmers to tailor their code to achieve maximum GPU performance. We develop analytical models to predict the performance of GPUs for computationally intensive tasks. We verify the proposed models using benchmark programs from the Nvidia CUDA GPU Computing SDK and also on multiple platforms including both Tesla and the latest Kepler GPUs.

1.4 Contributions of the Dissertation

The *contributions* of this dissertation are summarized as follows:

- Develop a mathematical model to estimate upper- and lower-bounds of CPU availability for single- and multi-core architectures.

- Perform empirical evaluation in both Windows systems with Java language and Linux systems with C language to validate the accuracy. An average CPU availability model is introduced as a result of the empirical studies.

- Develop a theoretical model to measure the execution efficiency of concurrent threads while the memory availability is uncertain.
- Combine the CPU and memory model non-trivially to derive a new composite prediction model and evaluate empirically.
- Introduce two task assignment policies that are capable of dispatching tasks to distributed multi-core compute nodes intelligently by utilizing composite prediction and CPU usage models.
- Propose a technique and introduce a model to identify candidate combinations of parameters for efficient usage of the GPU device to obtain optimal performance (for heterogeneous GPU devices).

1.5 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 discusses the basics and relevant work in CPU, memory, task assignment, and GPU areas. Chapter 3 presents the theoretical CPU availability prediction model and conducts empirical work in JVM running under Windows to observe the accuracy. In Chapter 4, we perform empirical work in Linux environment for validating the CPU model utility for a different platform. This chapter also introduces a compact lower-bound model (useful for the Linux systems running C programming language) and an average CPU availability model. Chapter 5 presents the memory model with empirical work followed by the composite CPU/Memory model. Two task assignment policies, Task Assignment using Status Measurements (TASM) and Task Assignment using Status Estimates (TASE), are introduced in Chapter 6 and empirically verified with real-world benchmark programs. The empirical results are then compared with the popular TORQUE system. In Chapter 7, we present a mathematical model to measure the performance of compute intensive tasks for GPUs. The empirical work is conducted for two different GPU architectures keeping heterogeneity in mind. Finally, Chapter 8 incorporates concluding remarks and future research directions.

Chapter 2

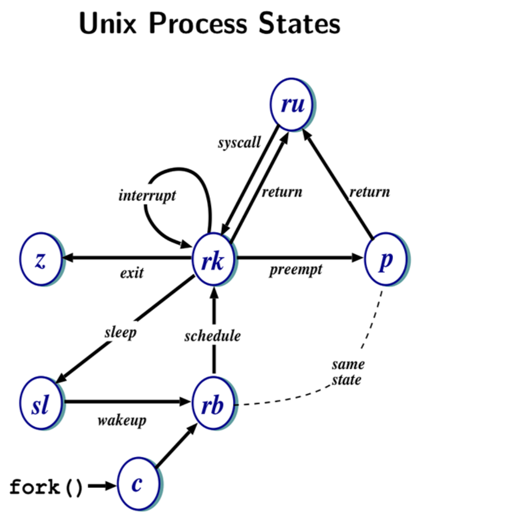
Literature Review

It is necessary to have accurate models for estimating resources of compute nodes in parallel and distributed systems because of the dynamic nature of computer systems and their workload. In this chapter, a review of different Linux schedulers (versions 2.4 and 2.6^{+}), resource availability prediction approaches, and Graphical Processing Unit (GPU) basics and programming model are provided to help the reader better understand context and prior research relevant to this dissertation.

2.1 Linux Scheduling at a Glance

The Linux scheduler makes it possible to execute multiple programs at the same time, thus sharing the CPU with users of varying needs. An important goal of a scheduler is to allocate CPU time slices efficiently while providing a responsive user experience. The scheduler can also be faced with such conflicting goals as minimizing response times for critical real-time tasks while maximizing overall CPU utilization [39]. An understanding of the basic ideas of Linux 2.4 and 2.6 and newer schedulers is necessary to clearly view the proposed models capable of predicting the CPU availability on a Linux workstation.

Processes are created by a parent process during execution. Usually the child processes are created to execute a binary from an existing process, with the system



ru	=	running (user-mode)	rk	=	running (kernel-mode)
z	=	zombie	p	=	pre-empted
sl	=	sleeping	rb	=	runnable
c	=	created			

Figure 2.1: Various states of Unix Processes [34].

call *fork()*. Processes are usually created to run through a shell or terminal. In this case the shell is the parent process that execute child processes. In Unix / Linux type systems each process has a parent process except the parent called init [34]. Figure 2.1 shows various states of new processes in the Unix environment. Almost all the activities make use of a system of processes to carry out their tasks. Active processes are placed in an array called a run-queue. The run queue may contain priority values for each process, which will be used by the scheduler to determine which process to run next. To ensure each program has a fair share of resources, each one is run for some time period (quantum) before it is paused and placed back into the run queue. When a program is stopped to let another run, the program with the highest priority in the run queue is then allowed to execute. Processes are also removed from the run queue when they are put to sleep, are waiting on a resource to become available, or have been terminated. When a processes is preempted, it enters in the state p. The preempted process is also runnable and shown by a dotted line between the states.

2.1.1 Linux Scheduler 2.4

The 2.4 scheduler divides the processor time into epochs or rounds, periods of time in which each process in the system is allowed to use its scheduling time slice. The length of these time slices is always computed at the beginning of the epochs so that it can consider how much CPU was utilized by each process in the last epoch. To compute the time slice length for a process, the base time slice is modified, taking into account its CPU requirements. Each process has a counter to count the clock ticks remaining in its current time slice. If the process is CPU-bound and uses its whole time slice, the counter value is 0 at the end of this slice and the next time slice will be the base time slice. But, if the process does not use its whole time slice because it is I/O-bound, the next time slice for this process will be longer; exactly the counter divided by two is added to the base time slice [1]. This scheduler is based on relatively easy ideas, works well, and has been widely used. However, it is an $O(n)$ algorithm (the length of each process time slice has to be computed at the beginning of each epoch) with scalability limitations [13]. Furthermore, it has been observed that the average time slice assigned to processes is about 210 ms, which is too high for many environments and associated domain [1].

2.1.2 Linux Scheduler 2.6^{+}

The 2.6 and newer schedulers are based on the same basic ideas found in version 2.4, but it uses the run queue structure and the priority arrays to achieve an $O(1)$ complexity. The new $O(1)$ scheduler of the Linux kernel can schedule processes within a constant amount of time, regardless of how many processes are running on the operating system. All processes have a default static priority. Each CPU has a run-queue made up of 140 priority lists that are serviced in First In First Out

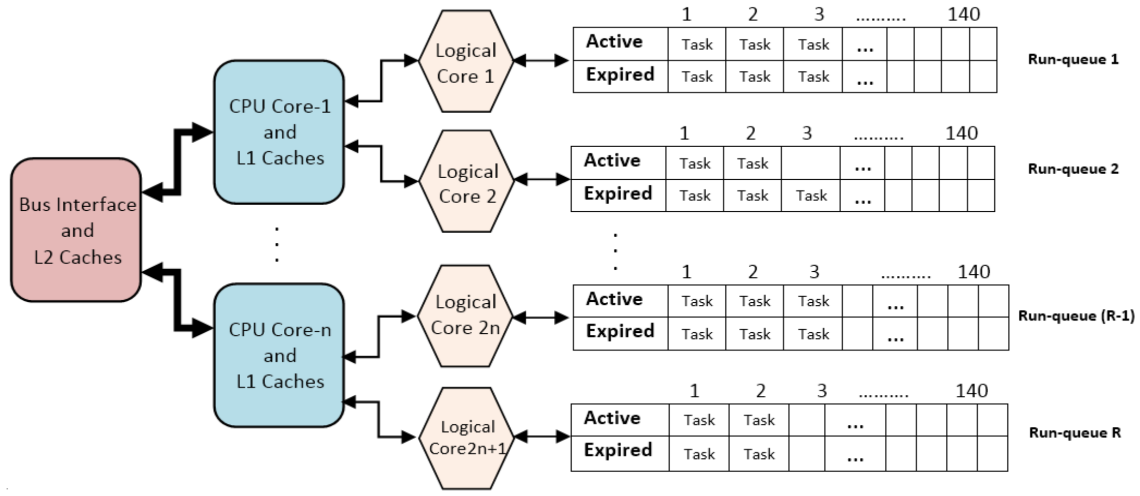


Figure 2.2: Typical run-queue of Unix systems containing 140 priority lists served in FIFO order.

(FIFO) order. Each task has a time slice (length depends on the priority level) that determines how much time it's permitted to execute. The first 100 priority lists of the run-queue are reserved for *real-time tasks*, and the last 40 are used for *user tasks* [36].

Figure 2.2 shows the logical run-queue of Unix systems. Depending on the process variation (CPU versus I/O bound), the scheduler *dynamically modifies* this measured value to assign a priority penalty to CPU-bound processes and boost I/O-bound processes. This dynamic priority scheme is controlled with the sleep_avg variable for each process. When an I/O-bound process is awakened from a sleep interval, its total sleep time is added to this variable. In addition, when a process leaves the processor, the time it has been executing (on it) is subtracted from this variable. The higher the sleep_avg value, the higher the dynamic priority will be. A process with higher priority will get larger time slices compared with a lower priority process. In addition, the time slice of a process is computed with its priority value, always maintaining its length between pre-defined minimum and maximum values [1], [14].

The Completely Fair Scheduler (CFS) was merged into the 2.6.23 release of the Linux kernel and is the default scheduler. It handles CPU resource allocation for executing processes, and aims to maximize overall CPU utilization while also maximizing interactive performance [20]. In contrast to the previous $O(1)$ scheduler used in older Linux 2.6 kernels, the CFS scheduler implementation is not based on run queues. Instead, a red-black tree implements a *timeline* of future task execution. Additionally, the scheduler uses nanosecond granularity accounting, the atomic units by which an individual process' share of the CPU was allocated (thus making redundant the previous notion of time-slices). This precise knowledge also means that no specific heuristics are required to determine the interactivity of a process [20].

Like the old $O(1)$ scheduler, CFS uses a concept called *sleeper fairness*, which considers sleeping or waiting tasks equivalent to those on the run-queue. This means that interactive tasks which spend most of their time waiting for user input or other events get a comparable share of CPU time when they need it [20].

2.2 CPU Availability Predictions

Research that is closely related to this dissertation falls under two different prediction models: those based on static CPU availability prediction and those based on dynamic CPU assignment prediction models. These two approaches have a significant effect in areas such as dynamic load balancing, scalability analysis or parallel systems modeling.

Two new response time prediction models were introduced by Beltrań et al. [46]. Both of these models are static prediction models. The first one is a mixed model based on two widely used models, CPU availability and Round Robin models. The second one, called Response Time Prediction (RTP) model, is a new model based on a detailed study of different kinds of tasks and their CPU resource requirements. A large set of tests shows the predictive power of these models. The RTP model exhibits an error of less than 2% in all the empirical cases considered [46].

In a highly influential paper, Kunz [28] showed the influence of workload descriptors on the load balancing performance and concluded that the best workload descriptors is the number of tasks in the run queue. In the paper by Zohu [54], the CPU queue length is used as an indication of processor load. The number of tasks in the run queue is presented as the basis of a good load indicator, but an improvement is proposed by averaging this length over a period of one to four seconds [46]. Because of the dynamic nature of current computer systems and their workloads, making such predictions is not simple as workload in a system can vary drastically in a short interval of time. These simple models can incur large errors in particularly when there are heterogeneous tasks and resource requirements.

Almost all of the prediction models are focused on availability or workload predictions (inversely related) or on process response time predictions in mono processor systems [31]. In general, there are four major approaches to obtain performance predictions in a computer system: code analysis, benchmarking, statistical prediction, and analytical modeling. Low level tools for counting and studying program instructions and their dependencies has been proposed and implemented in papers by Chandra [70] and Park [71] using code analysis to predict response times. These approaches are often architecture dependent (therefore, they generally are not portable) and applicable only to a specific class of tasks. Benchmarking techniques use exhaustive code profiling, as in [70], and they are usually complemented with statistical approaches, based on predicting the future from past observations. One example of statistical prediction illustrated by Wolski in [31] focused on making short and medium-term predictions of CPU availability on time-shared Unix systems. Mehra and Wah [41] presented an analytical method based on neural networks for automatically learning to predict CPU load, directly related to task response times.

Federova et. al [12] also worked on operating system scheduling on heterogeneous core systems. They proposed thread-to-core assignment algorithms that optimize performance and demonstrate the need for balanced core assignment. The paper makes the case that thread schedulers for multi-core systems in a heterogeneous environment should target the following objectives: optimal performance, core assignment balance, response time, and fairness. In addition, Federova [6] introduced a practical new method for estimating performance degradation on multi-core processors, and it's application to workloads of clusters nodes.

Finally, Dinda [30] has evaluated linear models for processor load prediction and implemented a system that could predict the running time of a compute-bound task on a host. Analytical modeling tries to give a high-level explanation of the effects of different parameters, such as the scheduling policy, the number of tasks in the run queue, the task arrival rate, and the service time, on the process response times. Most of these methodologies are based on queuing models, due to their simplicity, and almost all conclude that the best workload descriptor, often used to make the performance predictions, is the number of tasks in the run queue.

2.2.1 Short and Medium term Availability Predictions

Next relevant work on CPU availability of time-shared Unix systems by Wolski [31] and Jones [36] have focused on the problem of making short and medium term predictions. The accuracy with which availability can be measured has been evaluated using Unix load average, the Unix utility *vmstat*, and the Network Weather Service CPU sensor that uses both. The problem of predicting available CPU performance on Unix time-shared systems has been examined for the purpose of building dynamic schedulers. In this regard, the contributions it makes are:

- An exposition of the measurement error associated with popular Unix load measurement facilities with respect to estimating the available CPU time a process can obtain.

- A study of one-step-ahead forecasting performance obtained by the Network Weather Service (a distributed, on-line performance forecasting system) when applied to CPU availability measurements.

- An analysis of this forecasting performance in terms of the autocorrelation present between consecutive measurements of CPU availability.

- Verification of this analysis through its application to longer-term forecasting of CPU availability.

The results presented by Wolski in [31] are somewhat surprising in that they demonstrate the possibility of making short and medium term predictions of available CPU performance despite the presence of long-range autocorrelation and potential self-similarity. The obtained predictions exhibited a mean absolute error of less than 10%, typically making them accurate enough for use in dynamic process scheduling.

From the data presented by Wolsk in [31], it shows short-term (10 seconds) and medium-term (5 minute) predictions of CPU availability (including all forms of error) can be obtained that are, on the average, between 5% and 12%. These results are encouraging in the context of process scheduling as researchers assume that CPU load vary to such a degree that it is difficult to make dynamic scheduling. The paper shows dynamic measurement and forecast error combined are small enough so that effective scheduling is possible.

2.2.2 Static Process Assignment Prediction Model (SPAP)

The SPAP model introduced by Beltrań et al. [1] assumes prior knowledge about the tasks in the run queue and the *unloaded CPU usage* for these processes. Tasks in the run queue can be CPU bound or I/O bound, or a mix of both, resulting in different processor requirement to complete all tasks. The paper by Beltrań et al. [1] carried out an empirical average-case study in two different extreme situations to

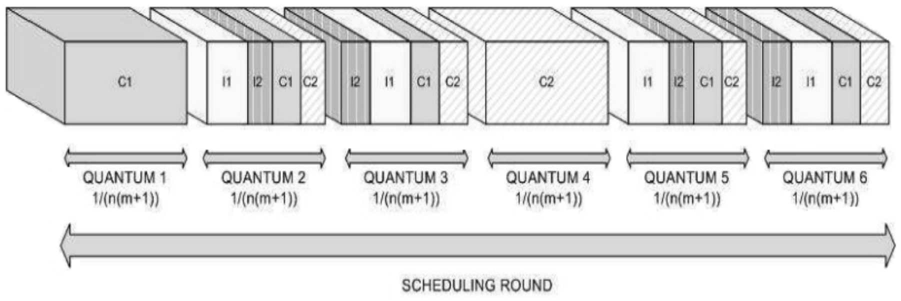


Figure 2.3: Two CPU bound ($n = 2$) and two I/O bound ($m = 2$) processes execution diagram with very low unloaded CPU usage values for I/O-bound processes [1]

propose the desired new model because the Linux scheduler behavior is different in these two situations. Consider a scenario where there are two CPU bound and two I/O bound processes in the run queue with very low CPU usage. The CPU usage, denoted by $\hat{U}$, is defined as the CPU time consumed by a process divided by it's overall execution time, both with the processor is unloaded (e.g., $\hat{U} = 0.2$). In this scenario, the scheduler behavior leads, on average, to the execution diagram shown in Figure 2.2. This diagram shows a complete scheduling round or epoch: a scheduling interval that is periodically repeated in the processor. The different blocks indicate the beginning and the end of the processor time slices or quantums, C1 and C2 are the CPU-bound processes, and I1 and I2 are the I/O-bound processes.

In Figure 2.2, observe that the scheduler assigns one time slice or quantum to every CPU bound process and these processes take advantage of all of the CPU time that is allocated to them. Quantum 1 is for the process C1 and quantum 4 is for the process C2. Similarly, the Linux scheduler allocates time slice to an I/O bound process for every CPU bound process in the run queue. For the process I1, quantums 2 and 5 have been assigned and for the process I2, quantums 3 and 6 have been assigned. Also from the figure, note that the I/O bound processes cannot take advantage of all their time slices. For example, the I1 process is assigned to the processor during quantum 2, but this process only uses a certain fraction of the CPU time available during this quantum time, illustrated by its unloaded CPU usage. The rest of this quantum is used by the rest of the processes in the run queue. The I2 process can take advantage of the processor during a fraction of the quantum time, again given by its unloaded CPU usage. In addition, the rest of the quantum is equally shared by the CPU-bound processes that are always ready to run in the CPU and are capable of taking advantage of all of the available processor time.

The same phenomenon can be observed in quantums 3, 5, and 6, in which I/O bound processes are assigned to the processor. On average, this behavior is coherent with the Linux scheduler mechanisms, without considering the context switching overhead, because it always rewards I/O-bound processes and punishes CPU-bound processes. It can also be observed from Figure 2.3 that the I/O bound processes are given higher priority for processing than CPU bound processes. As I/O bound processes cannot use the CPU all of this time, the CPU-bound processes steal from them for some fraction of their quantums. Figure 2.3 has helped summarize this behavior to derive the desired analytical model. Here, the total number of CPU-bound processes is defined by n. From Figure 2.2, one I/O-bound process P_i with $i = 1, 2, ..., m$, on average, is actually using the processor at a fraction of the total time:

$$\frac{n \times \hat{U}_i}{n + n \times \sum_{j=1}^{m} \hat{U}_i} = \frac{\hat{U}_i}{1 + \sum_{j=1}^{m} \hat{U}_j} \tag{2.1}$$

However, this process assigned more time to the processor, which is actually a fraction of the total time given by $1/(m + 1)$. The prediction model proposed in the papers ([1], [8]) are based on the concept they have termed *robbery*. As an I/O-bound process does not take advantage of all its correspondent quantums, the rest of the processes in the run queue can *steal* part of this time from this I/O-bound process. In the same way, it will *steal* time from the rest of the I/O-bound processes when it is possible. In addition, the CPU-bound processes will take advantage of all of their quantums and of all of the time they can steal from the rest of the I/O-bound processes. Therefore, the process P_j (with $j = 1, ..., m$ *and* $j \neq i$) can *steal* time from the fraction of this slice in which the CPU is idle. Therefore, the total CPU time that P_i can steal from other non-CPU bound processes are:

$$\frac{\hat{U}_i^2}{m+1} \times \sum_{j=1, j\neq i}^{m} (1 - \hat{U}_j) \tag{2.2}$$

The CPU time of an I/O-bound process is computed, taking into consideration the contributions of Eqs. 2.2 and 2.3, its own time slice, and the time stolen from other slices, respectively given by the two previous equations. The CPU-bound processes in the run queue get the equal amount of the remaining CPU time.

$$U_i = \begin{cases} \frac{\hat{U}_i}{1+\sum_{j=1}^{m} \hat{U}_j} + \frac{\hat{U}_i^2}{m+1} \times \sum_{j=1,j\neq i}^{m} (1 - \hat{U}_j) & \text{for all } \; i \in [1, m] \\ \frac{1-\sum_{j=1}^{m} \hat{U}_j}{n} \qquad \text{if } i = m+1, ..., m+n & \text{otherwise} \end{cases} \tag{2.3}$$

In Eq. 2.3, the first part of the equation computes the shared CPU usage of the I/O bound process in the run queue. The second part of the equation computes the usage of the CPU bound process in the run queue. Next, when I/O bound processes are very close to being CPU bound processes, i.e., $\hat{U}$ value is very close to 1, it leads to scheduling of I/O bound process only once in the scheduling round similarly as CPU bound processes [1]. The sharing of the processor time in the quantums of the I/O-bound processes is the same as in the previous case. Thus, it can be explained with the *robbery* concept (in quantum 2 for the I1 process and in quantum 3 for the I2 process).

This expression gives the general equation to obtain the fraction of CPU time that will correspond to any process in a time-shared environment. According to experimental results, some modifications can be made in order to improve the model performance. All of the I/O bound processes that have unloaded CPU usage values below 0.15 and the cases where all of the processes are CPU bound or near CPU bound have been separated to reduce the prediction error of the model. In the first case, it has been noted that the prediction error improves, considering $\hat{U} = U$ for all of the I/O-bound processes. For example, a process with an unloaded CPU usage of 0.05 will have a very similar value of CPU usage when sharing the processor with other processes because its decrease is practically negligible. In the second case, the RR model can be used because, when all of the processes in the run queue are CPU-bound or near CPU bound, this model obtains good prediction errors and guarantees a minimum CPU assignment of $n = 1$ for each process. Therefore, the definitive expressions for the static prediction model SPAP, denoted using α, is:

$$\alpha = max\left(\frac{1-\sum_{i=1}^{m} U_i}{n+1}, \frac{1}{n+m+1}\right) \tag{2.4}$$

with the expression of shared CPU usage is the following:

$$U_i = \begin{cases} \hat{U}_i & \text{if } \hat{U}_i < 0.15 \text{for all } i \\ \frac{\hat{U}_i}{1+\hat{U}_i\times(n-1)+\sum_{j=1}^{m}\hat{U}_j} + \frac{\hat{U}_i^2}{m+1+\hat{U}_i\times(n-1)} \times \sum_{j=1,j\neq i}^{m}(1-\hat{U}_j) & \text{otherwise} \end{cases} \tag{2.5}$$

All of the processes in the run queue can take advantage of their processor quantums. Various experiments ([1], [8]) have been conducted to demonstrate the limitations of RR scheduling when there are I/O bound processes in the run queue. When all the processes in the run queue are CPU bound (t100) or near CPU bound, the RR model achieves very low prediction errors but when there are processes with very low unloaded CPU usage, or I/O bound processes, the prediction error increases sharply. When there are I/O bound processes in the run queue the RR model is not capable of predicting what happens with the spare (unused) processing time which might increase the prediction error to around 30% which is unacceptable [1].

Also their experimental results shows that the prediction errors obtained with the SPAP model are nearly constant and always less than 4 percent. This new static model has very accurate behavior that is independent of the processor state and the run queue process CPU requirements. In some cases, the error obtained with the two models is exactly the same because, in certain situations, the SPAP model becomes the same expression that the RR model uses.

2.3 Memory Access Facts

When the required memory of processes is more than the amount of available memory, the resulting process execution performance can degrade sharply. As more processes are placed into the primary memory, due to demand paging, the chance of memory overrun increases. This situation (requiring additional memory for code and data) can trigger page swapping from primary memory to backing store (swap-out pages are selected from the existing processes) which can increase the process execution time immensely. Even if a new process gets 99% of required pages in primary memory, the pager still needs to swap out memory frames from existing processes to make room for the remaining 1% of pages.

To demonstrate the effect, let p be the probability of the page fault ($0 \leq p < 1.0$). It is desired to keep p to be close to zero. The effective access time, denoted by ea, is expressed in terms of memory access time, A_p, and page fault service time as [13]:

$ea = (1 - p) \times A_p + p \times page\, fault\, time$

With an average page-fault service time of 8.0 milliseconds and memory access time of 200 nanoseconds, the effective access time,

$ea = (1 - p) \times 200\, nanoseconds + p \times 8\, milliseconds$

$ea = 200 + 7,999,800 \times p\, nanoseconds$

That is, effective access time is directly proportional to the page-fault rate, p. If one page out of 1000 causes a page fault, the effective access time is 8.2 microseconds [13]. The system will be slowed down by a factor of 40 due to page fault service time. *Predicting the process execution performance when the available memory is less than required memory is important in making process assignment and scheduling decisions.* When the required memory by the processes is more than the amount of available memory, the resulting process execution performance can degrade sharply.

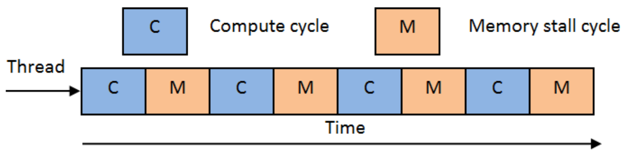


Figure 2.4: Memory stall situation [13] due to cache miss results in up to 50% stall time overhead.

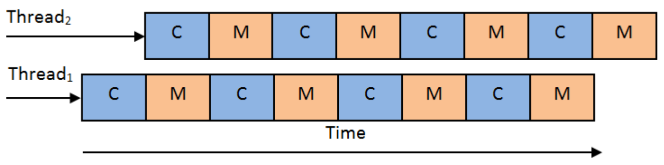


Figure 2.5: Multithreaded execution to hide the effect of memory stalls [13] to reduce access time.

When a processor accesses memory, it spends a significant amount of time waiting for the data to become available because of cache misses which may result in up to 50% of stall time shown in Figure 2.4. This situation generates huge overhead when the frequency of memory access increases. To overcome this situation, most of the recent hardware designs have implemented multi-threaded processor cores in which two or more hardware threads are assigned to each core [13]. Figure 2.5 shows that way, if one thread stalls while waiting for memory, the core can switch to another thread [13]. To maintain its own architectural state, each core has its independent register set and thus appears to the operating system to be a separate physical processor. From an operating system perspective, each hardware thread (when hyper-threading is enabled) appears as a logical processor that is available to run a software thread. Thus, on a dual-threaded, dual-core system, four logical processors are presented to the operating system. We have incorporated the effect of hyper-threading in our new CPU availability and memory models for accuracy.

2.4 Distributed Task Assignment Frameworks

A wide variety of approaches have been made for scheduling tasks in a distributed environment, many of these are application specific. A general (application independent) approach for arbitrary applications has been proposed in the paper by H. Dail [76] which focuses on a decoupling of the scheduling decisions from the application specific information by encapsulating the application characteristics in an analytical performance model and a data mapper.

Besides application independent approaches, there are several Grid scheduling techniques that have been developed for a specific class of applications and it is not necessarily straight forward to extend them to other applications or classes [10]. There are multiple *resource managers* for scheduling tasks in grid environments that are available. Most of the major production systems like TORQUE, Condor, PBS Pro, Sun Grid Engine (SGE), GridWay, gLite, and others uses a queuing system. In a queuing system the currently free resources are *matched* with the tasks with their resource requests. The systems mentioned above use various matching schemes and differ from the ones introduced in this dissertation in the sense that our matching scheme is based on the availability models presented in that paper.

A complex data structure is deployed in job schedule that maps jobs onto available compute nodes in time [21]. The TORQUE is a distributed resource manager providing control over batch jobs and distributed compute nodes. An extension of TORQUE scheduler allowing the use of planning (scheduling for the future) and optimization in grid environment has been proposed by Chlumský but yet to implement [21].

2.5 GPU Environment

A graphics processing unit (GPU) is a specialized graphics card designed to rapidly manipulate display and is often used in embedded systems, mobile phones, personal

Figure 2.6: An Nvidia Tesla Kepler 20 GPU card. [60]

computers, workstations, and game consoles [79]. Modern GPUs are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. It is becoming increasingly common to use a general purpose GPU as a modified form of stream processor. Furthermore, GPU-based high performance computers are starting to play a significant role in large-scale modeling. Three of the 10 most powerful supercomputers in the world take advantage of GPU acceleration [79].

Figure 2.6 shows the Nvidia Kepler 20 GPU card. Kepler is Nvidia's first micro-architecture to focus on energy efficiency, programmability, and performance. Kepler micro-architecture provides dynamic parallelism, an ability that provides kernels (functions are called kernels in GPU) the advantage to be able to dispatch other kernels. With the previous micro-architecture Fermi, only the CPU could dispatch a kernel, which incurs a certain amount of overhead by having to communicate back to the CPU. By giving kernels the ability to dispatch their own child kernels, Kepler20 can both save time by not having to go back to the CPU, and in the process free up the CPU to work on other tasks [80].

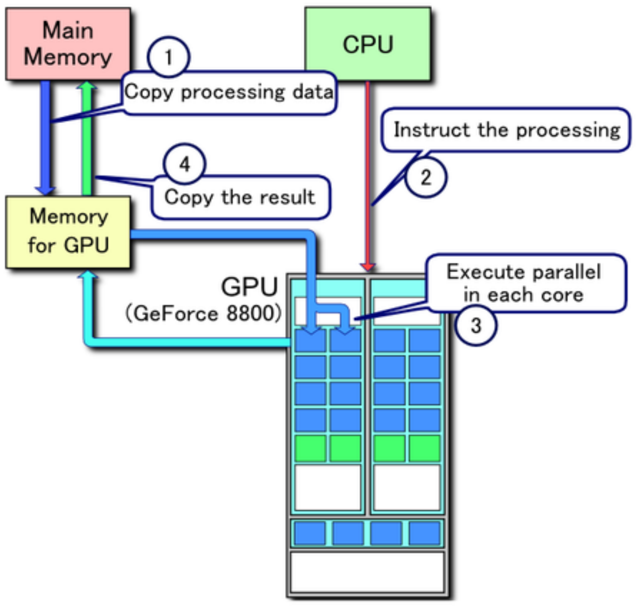


Figure 2.7: Typical processing flow on CUDA [81].

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by Nvidia and implemented for their GPUs [81]. Nvidia cards support API extensions to the C programming language such as Compute Unified Device Architecture (CUDA) and OpenCL. CUDA is specifically for Nvidia GPUs where as OpenCL is designed to work across a multitude of architectures. These technologies allow specified functions (known as kernels) from a normal C program to run on the GPU's stream processors. This makes C programs capable of taking advantage of a GPU's ability to operate on large matrices in parallel, while still making use of the CPU when appropriate. CUDA is also the first API to allow CPU-based applications to directly access the resources of a GPU for more general purpose computing without the limitations of using a graphics API [79].

Figure 2.7 shows the major steps of data copy from primary memory to GPU memory, parallel execution of kernels, and copying result back from GPU to primary memory. CUDA processing flow is given below:

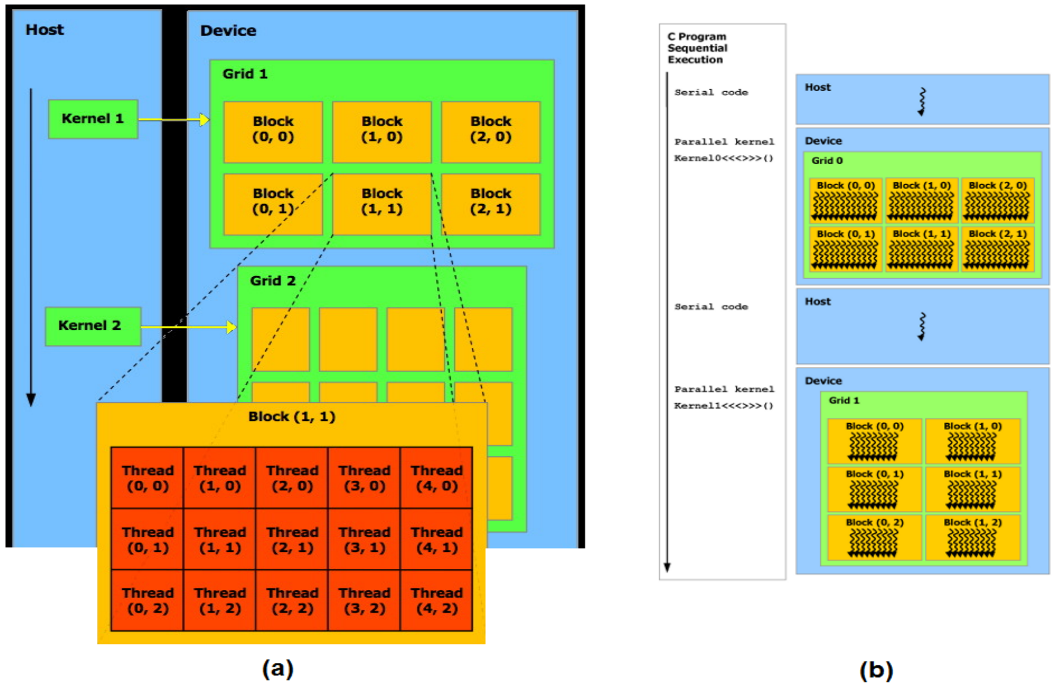


Figure 2.8: CUDA kernel execution: (a) thread arrangement in a grid of 2D blocks (b) serial execution of CPU portion and parallel execution of GPU kernel [87].

1. Copy data from main memory (Host) to GPU memory (Device)
2. CPU instructs the process to GPU
3. GPU execute parallel in each core
4. Copy the result from GPU memory (Device) to main memory (Host)

CUDA gives program developers direct access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs. Using CUDA, the GPUs can be used for general purpose processing (i.e., not exclusively graphics); this approach is known as GPGPU. Unlike CPUs, however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly [81].

The GPU is ideally used as a compute device to execute a portion of an application that

- has to be executed many times;
- can be isolated as a function; and
- works independently on different data.

Such a function can be compiled to run on the device. The resulting program is called a *Kernel*. A CUDA kernel is executed by an array of threads. All threads run the same code using Single Program, Multiple Data (SPMD) policy. Figure 2.8 (a) shows arrange of CUDA threads in a grid containing 2D blocks. Threads within the same block execute in SIMD fashion, whereas threads of different blocks can execute different portions of the code (SPMD: Single Program Multiple Data). Each thread has an ID that it uses to compute memory addresses and make control decisions. The batch of threads that executes a kernel is organized as a grid of thread blocks. Block can be one-, two- or three-dimensional arrays. Threads in different blocks cannot cooperate. Figure 2.8 (b) illustrates the parallel execution of kernel (function) that enables dramatic increases in computing performance by extracting the power of the GPU.

2.6 Summary

In this chapter, we have reviewed the basics of Linux scheduler (2.4 and 2.6^{+}), relevant work in predicting CPU availability, various distributed task assignment approaches, and GPU architecture and programming model. This review of prior related work serves as a foundation upon which the work of this dissertation is built. Specifically, in Chapter 3 and 4 we have presented the theoretical CPU availability prediction model and conducted empirical work in both Windows and Linux multi-core systems. In Chapter 5, we have introduced a memory model and derived composite CPU/Memory model non-trivially followed by extensive empirical work. Two task assignment policies, Task Assignment using Status Measurement (TASM) and Task Assignment using Status Estimates (TASE), are introduced in Chapter 6 and empirically verified with real-world benchmark programs. Finally, in Chapter 7, we have presented a mathematical model to measure the performance of compute intensive tasks for GPUs.

Chapter 3

CPU Availability Model in JVM

Techniques for predicting the availability of CPU resources associated with the execution of multiple concurrent Java threads on a multi-core architecture are introduced. Prediction of CPU availability is important in the context of making thread assignment and scheduling decisions. Theoretically derived upper and lower bound formulas for estimating CPU availability are introduced. The bounds are necessary because the actual CPU availability changes based on the order of execution of threads in the batch. Relevant experimental studies and statistical analysis are performed to validate the theoretical bounds and provide a basis for an empirical model for predicting CPU availability. To facilitate scientific and controlled empirical evaluation, synthetically generated threads are employed that are parameterized by their unloaded CPU usage factor, defined as the fraction of time a thread spends utilizing CPU resources on an unloaded system.

3.1 Introduction

Predicting the availability of CPU resources when the number of threads assigned to the processor exceeds the number of cores is important in making thread assignment and scheduling decisions [1]. Precise values of CPU availability are difficult to predict because of a dependence on many factors, including context switching overhead, CPU usage requirements of the threads, the degree of interleaving of the timing of the CPU requirements of the threads, and the characteristics of the thread scheduler of the underlying operating system (or Java Virtual Machine). Due to the complex nature of the execution environment, an empirical approach is employed to evaluate proposed CPU availability prediction models and formulas.

In this chapter, an extensive collection of empirical measurements taken from both single- and multi-core processors provide a basis for validating proposed analytical models for estimating CPU availability. The proposed models are theoretically derived upper and lower bound formulas for CPU availability. Confidence interval and moving average statistics from measured CPU availability (from different empirical case studies) validate the utility of these theoretical models. Case studies for a single core machine involve spawning 2, 3 and 4 concurrent threads with randomly selected CPU usage factors. Each case study includes a collection of about $2,000$ sample executions. For a quad core machine, similar case studies are conducted involving 8, 12 and 16 concurrent threads, again with randomly selected CPU usage factors for the threads.

An interesting outcome of the empirical case studies is that the variation in measured CPU availability is very low whenever the sum of the threads' CPU usage factors is either relatively low, or relatively high. Thus, prediction of CPU availability is quite accurate when the CPU is either lightly or heavily loaded. When the total CPU loading (i.e., sum of all threads' CPU usage factors) is moderate, the realized availability of CPU resources has more variation and thus is less predictable. However, even the largest variation in CPU availability measured was typically no more than 20%, with a 90% confidence. The empirical studies also show that, as the number of concurrent threads increases, the context switching overhead degrades the performance of thread execution; resulting in a slightly wider gap between the theoretically derived upper bound and measured availability values. In both single- and multi-core cases, when the number of threads is above the number of CPU cores, the performance of thread execution is predicted reasonably well by the theoretical upper bound formula for CPU availability.

The rest of the chapter is organized in the following manner. Section 3.2 discusses relevant background related to the execution of Java threads, and motivates the importance of predicting CPU availability. Section 3.3 introduces the thread execution framework assumed in this section and verifies the assumed model. Section 3.5 provides the theoretical derivations of upper- and lower-bound formulas for CPU availability model. Section 3.6 presents the empirical studies including benchmarking, case study measurements for single- and multi-core CPUs, and statistical analysis of the results. Finally, Section 3.7 provides the summary of contributions and achievements of this chapter.

3.2 Execution of Concurrent Java Threads

The primary focus of this section is to estimate the CPU resource availability of a Java Virtual Machine (JVM) executing concurrent Java threads. The concept of a Java monitor [4] is useful for describing how threads are executed by a JVM. A graphical depiction of such a monitor is shown in Figure 3.1 (derived from [4]), which contains three major sections. In the center, the large circular area represents the owner, which contains two active threads, represented by the small shaded circles. The number of threads the owner can contain (concurrently) is bounded by the number of CPU cores. At the upper left, the rectangular area represents the entry set. At the lower right, the other rectangular area represents the wait set. Waiting or suspended threads are illustrated as striped circles.

Figure 3.1 also shows several labeled *doors* that threads must pass through to interact with the monitor. In the figure, one thread is suspended in the entry set and one thread is suspended in the wait set. These threads will remain where they are until one of the active threads releases its position in the monitor. An active thread can release the monitor in either of two ways: it can complete the monitor

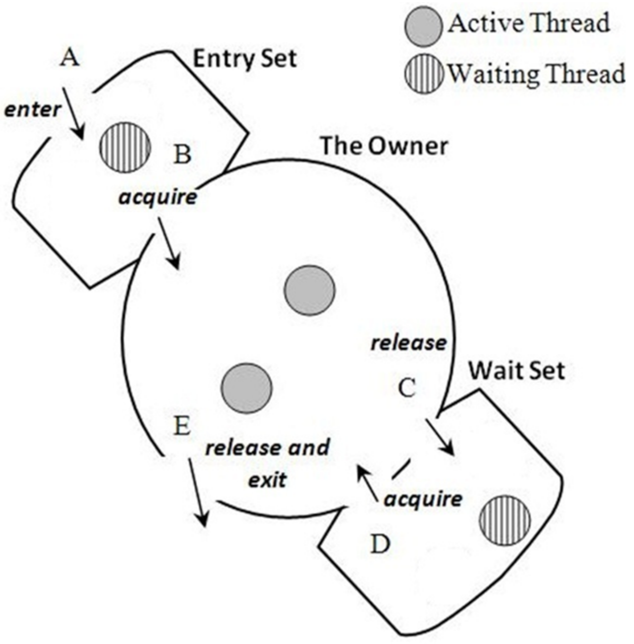


Figure 3.1: A typical Java Monitor executing concurrent threads.

region it is executing or it can execute a wait command. If it completes the monitor region, it exits the monitor via the door labeled E. If it executes a wait command, it releases the monitor and passes through door labeled C, the door into the wait set.

To illustrate issues associated with predicting CPU availability, assume the monitor of Figure 3.1 is associated with a dual-core JVM. From the figure, there are a total of four threads assigned to the JVM; two are currently in a waiting state and two are currently active. If all four threads are CPU-bound, meaning that their unloaded CPU usage factor is 100%, then it is clear that the availability of the CPU resources (i.e., both CPU cores) will be about 50%. If, on the other hand, some threads are I/O-bound (e.g., having CPU usage factors less than say 25%) then predicting CPU availability is not as straightforward. In general, the realized CPU availability depends upon the scheduling scheme employed by the JVM (and/or the underlying OS) for transitioning threads between the active and waiting states.

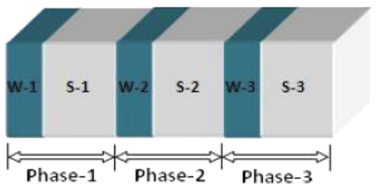


Figure 3.2: Multiple work and idle phases of a thread.

In the next section, an analytical framework is developed for estimating CPU availability associated with executing concurrent threads on a multi-core JVM. The primary contribution of the section is the derivation of upper and lower bound formulas for CPU availability.

3.3 Analytical Framework for CPU Availability Model

In this section, an analytical framework is developed for estimating the CPU availability associated with executing concurrent threads on a multi-core processor. A thread is modeled by a series of alternating work and sleep phases. For the purposes of this study, the work portion of a phase is CPU-bound and requires a fixed amount of computational work (i.e., CPU cycles). The sleep portion of a phase does not consume CPU cycles, and its length relative to that of the work portion is used to define the CPU load usage factor for a thread. A thread may be in idle/sleep phase because of I/O or any other types of interruption [7]. Figure 3.2 shows three work-sleep phases of a thread.

In the framework described here, a thread in the work portion of a phase will remain in the work portion until it has consumed enough CPU cycles to complete the allotted work of that portion of computation. After completing the work phase, the thread then enters in the idle/sleep portion where it stays (does not consume CPU cycles) for a specific amount of time defined by the CPU usage factor. Each

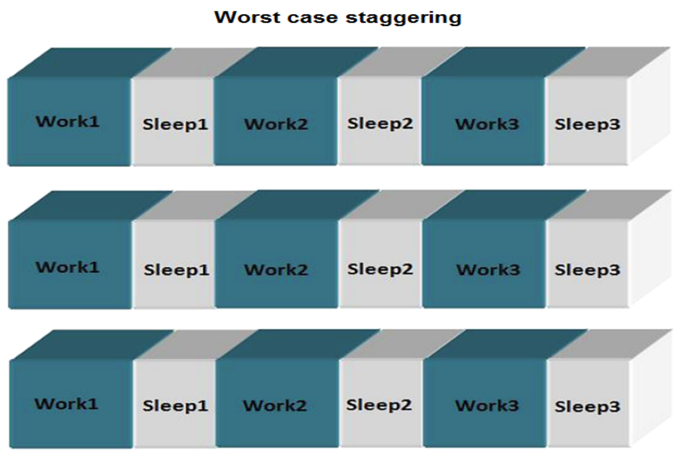


Figure 3.3: Three concurrent threads in a single-core machine with identical work-sleep phases and CPU usage factors with *contending* work portions.

thread is parameterized by a CPU usage factor. The *CPU usage factor* is defined as the time required to complete the work portion of a work-sleep phase on an unloaded CPU, divided by the total time of a work-sleep phase. A thread having zero sleep length has a CPU usage factor of 100%, which is also called a *CPU-bound thread*. Sleep phase length and the total amount of computational work to accomplish is computed based on the CPU usage factor. Sleep phase length of a thread is relative to that of the work portion based on CPU load factor.

When multiple threads are spawned concurrently, the JVM runs those threads employing a time sharing technique (refer to the discussion of the Java Monitor in Section 3.2). The CPU availability (and performance) will be degraded when the work phase of all threads overlap each other in time. Figure 3.3 depicts a scenario where 3 threads with identical work-sleep phases are executed in a single-core execution environment.

Each thread gets a maximum of 1/3 of the available CPU resource during the work portions of their phases, resulting in 3-times wider work portion execution time than would be the case for a single thread scenario. Alternatively, if the work portions of these three threads are staggered to where there is no overlap, then

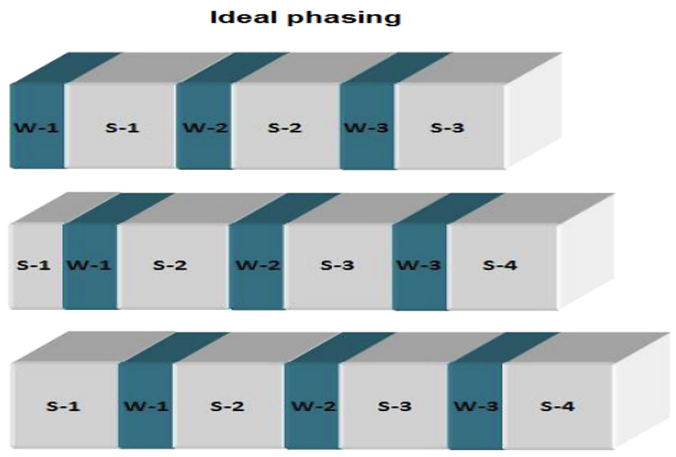


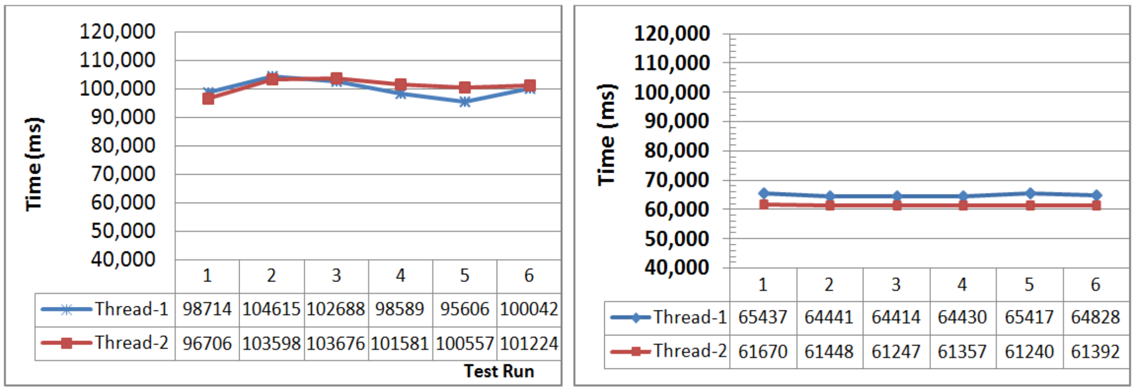
Figure 3.4: Three concurrent threads in a single-core machine with identical work-sleep phases and CPU usage factors with *interleaved* work portions.

there is no contention for the CPU resource and the CPU availability is essentially 100%, as shown in Figure 3.4. That is, all the work phases of concurrent threads are separated in time so that each thread can get the full usage of the CPU the moment it is first needed by each thread. The ideal phasing illustrated in Figure 3.4 requires that the work of any two threads can be accomplished within the time of one sleep portion of a phase.

In the proposed framework, each thread is parameterized by a CPU usage factor and a total amount of computational work to accomplish. The CPU usage factor is defined as the time required to complete the work portion of a work-sleep phase on an unloaded CPU, divided by the total time of a work-sleep phase. A thread having zero sleep length has a CPU usage factor of 100%, which is also called a CPU-bound thread.

3.4 Run-time Verification for Concurrent Threads

The purpose of this section is to empirically verify the correctness of the assumed thread execution framework of Section 3.3. This verification is necessary because the



	1	2	3	4	5	6
Thread-1	98714	104615	102688	98589	95606	100042
Thread-2	96706	103598	103676	101581	100557	101224

(a)

	1	2	3	4	5	6
Thread-1	65437	64441	64414	64430	65417	64828
Thread-2	61670	61448	61247	61357	61240	61392

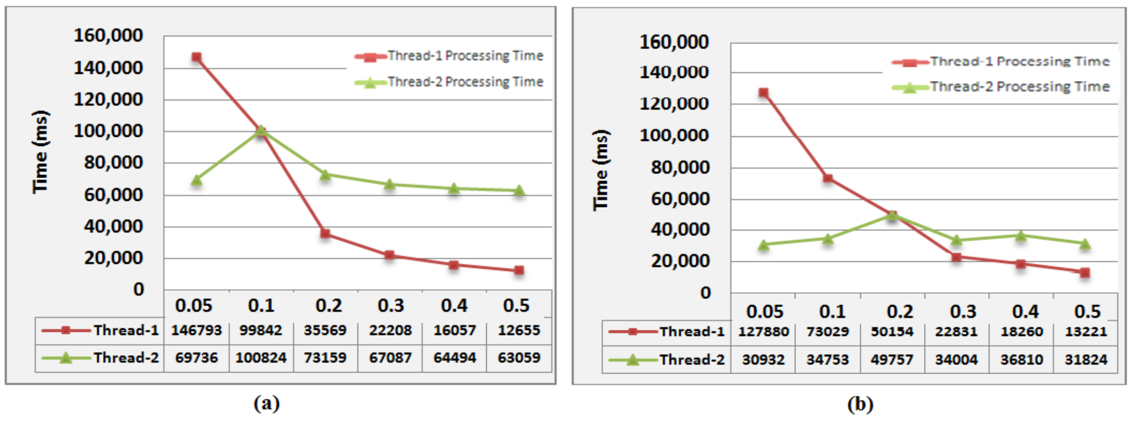
(b)

Figure 3.5: Two concurrent threads in a single-core machine (a) With *contending* (identical work-sleep phases and CPU usage factors) work portions. (b) With *interleaved* work portions.

assumed thread execution model plays a vital role in the CPU availability prediction model for designing and implementing synthetic benchmark program. Besides, this section discusses the surroundings of the problem associated with executing concurrent threads with the same CPU load on JVM.

A rapid programming approach is taken to verify the execution time when two or more concurrent threads are spawned in JVM with the same CPU load. The system that is used for handling the test cases is an Intel Xeon CPU E5540 with 2.53GHz clock speed, $1,333$ MHz bus speed and 4 GB of RAM. Experimental programs are implemented in Java and executed using JDK 1.6. Threads are given 2.0×10^8 units of synthetic work load which needs to be accomplished in 50 work-sleep cycles. Threads need to accomplish 4.0×10^6 units of synthetic work in each work phase. Ideal execution time (no CPU contention) for a thread to complete total synthetic work is measures as $58,661$ ms.

Two concurrent threads are spawned in a single core machine, running JVM, with the same amount of synthetic work (2.0×10^8 units) and CPU load (0.1). Figure 3.5 (a) shows the running time of 5 independent test runs. In this figure, the horizontal axis represents test run number and vertical axis represents total run-time. It can be



(a)

	0.05	0.1	0.2	0.3	0.4	0.5
Thread-1	146793	99842	35569	22208	16057	12655
Thread-2	69736	100824	73159	67087	64494	63059

(b)

	0.05	0.1	0.2	0.3	0.4	0.5
Thread-1	127880	73029	50154	22831	18260	13221
Thread-2	30932	34753	49757	34004	36810	31824

Figure 3.6: Two concurrent threads in a single-core machine (a) A major increase in run-time of thread-2 can be observed when both threads have same CPU load of 0.1 (10%). (b) Similar effect for thread-2 with CPU load of 0.2 (20%).

observed from Figure 3.5 (a) that the average run-time for both concurrent threads have sharply increased to $99,842$ ms for thread-1 and $100,824$ ms for thread-2. It has been experimentally observed that if those two threads have exact same CPU load, the thread run-time increased sharply. As the work phases of both threads staggered on top of each other, each thread gets a maximum of $1/2$ CPU attention which eventually doubles the work time. Moreover, both threads are working at the same time and idle at the same time which minimizes the CPU utilization as well.

To overcome this situation, phase shifting concept introduced in Section 3.3 is used to empirically verify the usefulness of the technique. In this experiment, two synthetic threads are spawned in the same single-core machine with same work load but a slight CPU load ($\epsilon = 0.005$) gap to introduce phase shifting. Here, thread-1 is assigned 0.095 CPU load and thread-2 is assigned 0.1 CPU load. That is, the CPU load gap between these two threads is 0.005 or 0.5%. It can be observed from Figure 3.5 (b) that the the run-time for thread-1 has dropped from $99,842$ ms to $64,850$ ms (appox.) and for thread-2 it's $100,824$ ms down to $61,250$ ms (appox.). The CPU load difference of 0.005 between threads has helped to shift work phases and sharply reduce total run-time.

A Second set of test cases are carried out to further analyze the above findings involving two concurrent threads in a single-core machine. In these test cases, the CPU load (utilization) of thread-1 is varied from 0.05 to 0.5 (5% to 50%) where as the CPU load of thread-2 remains constant 0.1 for all cases. These test case results should provide a clear visualization of CPU resource contension between two threads. In Figure 3.6 (a), the horizontal axis represents *CPU load of thread-1* and the vertical axis represents *thread execution time in milliseconds*. The test results show that the execution time of thread-2 is generally closer to the benchmark run-time except the case where thread-1 and thread-2 has the same CPU load of 0.1. That is, when the CPU load of thread-1 reaches the same 0.1 like thread-2, the run-time of thread-2 increases sharply from average measured run-time of $64,500$ ms to $100,824$ ms. For all other cases, work times of thread-2 are acceptably closer to the average run-time.

In next test cases, CPU load of the first thread is varied from 0.05 to 0.5 units (like previous one) but the CPU load of the second thread is changed to 0.2 unit (remains constant). Figure 3.6 (b) also shows a similar finding where the run-time of the second thread is very close to average run-time of thread-2 except the case where CPU load of both threads are 0.2. Average run-time for thread-2 with CPU load of 0.2 is $33,750$ ms. When thread-1 and thread-2 has same 0.2 CPU load, the run-time of thread-1 increases to $50,154$ ms and thread-2 reaches to $49,757$ ms. It can also be observed from Figure 3.6 (a) that when the CPU load of thread-1 was 0.2, the run-time was $35,569$ ms but in Figure 3.6 (b), it has increased to $50,154$ ms which is approximately 70.91% increase in run-time. Similar case studies have been conducted considering all possibilities by employing different CPU load for thread-2. It is found for all the cases that when both threads have the same CPU load, the run-time increases noticeably. These observations suggests that when work phases of threads are in contention, the performance of both threads degrades significantly.

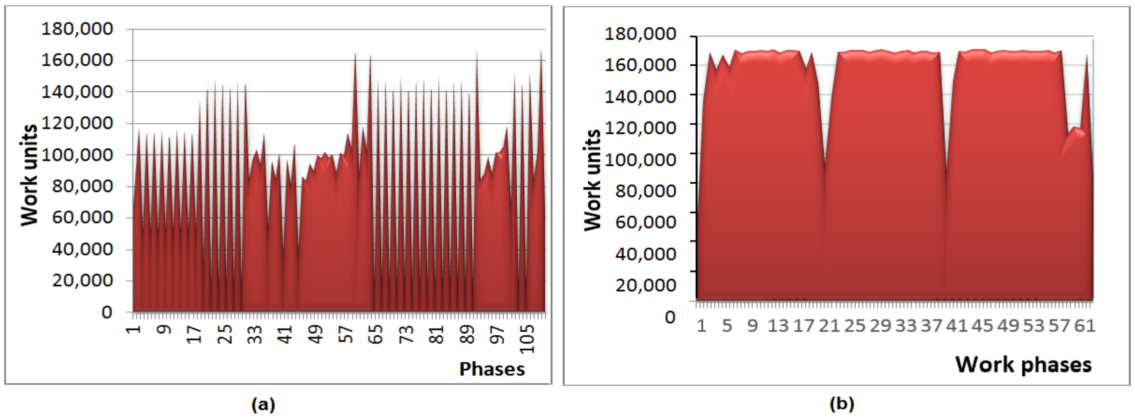


Figure 3.7: Variable amount of work accomplished by concurrent threads (a) When threads have *contending* work portions (b) When threads have *interleaved* work portions.

Figure 3.7 (a) shows the variable amount of work accomplishment when work phases of threads overlap. The amount of synthetic work that has been accomplished is approximately 81K units. Here, two threads are spawned concurrently with the same amount of CPU load and synthetic work. As these threads are CPU bound and starts with the work phase, the available CPU is shared among the concurrent threads. The work accomplishment varies based on the degree of work phase overlap among threads. If threads are fully overlapped, then the work accomplishment can drop as low as 18K units. When threads are interleaved, the work accomplishment can be as high as 162K units.

Figure 3.7 (b) illustrates the work accomplishment of the concurrent threads when phase shifting is utilized. The work accomplishment by a thread in each work phase is 158.5K units on average compared with the previous scenario where the average work accomplishment per phase was only 81K units (shown in Figure 3.7 (a)). Due to this improved approach through phase shifting, it took only 63 work phases to accomplish the total synthetic work instead of 110 phases (shown in Figure 3.7 (a)). Additionally, the context switching overhead has resulted in poor work accomplishment shown in the previous scenario. Figure 3.7 (b) also depicts that as

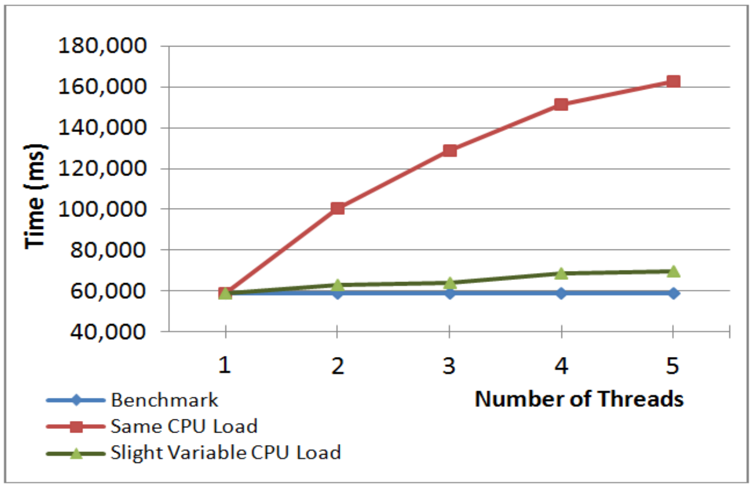


Figure 3.8: Run-time comparison of multiple concurrent threads in a single-core machine running JVM.

the work phase overlaps in cycle 1 and 2, 21 and 22, 40 and 41, and 59 to 61, the work accomplishment drops to around 98K units. For all other cases, work phases are mostly interleaved which enables them to get full attention of the CPU to achieve maximum amount of work.

Finally, Figure 3.8 shows a complete run-time scenarios for concurrent threads with contending and without contending work phases along with ideal execution time. These independent test cases includes 1, 2, 3, 4, and 5 concurrent threads spawned in a single-core machine. All test cases are conducted independently and run-time are measured for relative comparison. It can be observed from Figure 3.8 that the utilization of phase shifting through slight variation (only 0.5%) in CPU load has allowed threads to complete more work in each phase and complete the assigned work in a shorter period of time. This finding clearly suggests that the phase shifting is very efictive when concurent threads are spawned in the same machine to avoid CPU contention among threads.

3.5 CPU Availability Model Bounds

CPU availability model consists of upper- and lower-bounds. Most of the existing model provides one predicted availability value which might be useful for very short period of time. Making such predictions is complicated due to the dynamic nature of the system and its workload, which can vary drastically in a short span of time [1]. Besides, the actual CPU availability changes based on the order of execution of threads in the batch as well. Thus, the bound technique is necessary for distributed task scheduler. The bounds also provides best case and worst case scenario (deadline can be derived) which might be useful for many real-world applications. For example, when a task has a hard deadline, the model can be utilized to determine whether it is even possible to meet the deadline. Knowing the best and worst case will help to schedule tasks depending on deadlines as well.

The primary contribution of this section is the derivation of upper-and lower-bound formulas for CPU availability. The results presented in Section 3.3 shows that n running threads having similar work-sleep phases often result in $1/n$ CPU availability in a single-core machine. In such cases, the work phases get n times longer due to CPU contention among running threads, depicted in Figure 3.3. Alternatively, if the work portions of these three threads are staggered (interleaved) to where there is no overlap, by phase shifting, then there is no contention for the CPU resource and the CPU availability is essentially 100%, as shown in Figure 3.4. That is, all the work phases of concurrent threads are separated in time, by shifting the work phases, so that each thread can get the full usage of the CPU the moment it is first needed by each thread. The ideal phasing illustrated in Figure 3.4 requires that the work of any two threads can be accomplished within the time of one sleep portion of a phase.

3.5.1 Assumptions

The following are the assumptions for the CPU availability model:

- A batch of threads are spawned concurrently in a single- or multi-core system.
- The single- and multi-core systems in which threads will be spawned are dedicated, meaning they are not loaded with other threads.
- There are no inter-thread communication or message passing among threads.
- Overhead related to the operating system's real-time process execution is negligible.
- The CPU requirement of each thread is known, which is used to estimate the aggregate CPU load for the batch of threads.
- The CPU utilization is statistically "stationary" over time.

The model depends on the above specified assumptions. If these assumptions are not specified, then the model may produce prediction errors or may not be completely viable.

3.5.2 Bounds for Single-core Machines

For a single core machine, the following formulas define upper and lower bounds for CPU availability. Here, n is the number of threads assigned to the single core machine and L is the aggregate loading factor, defined as the sum of CPU usage factors of all n threads. The upper bound formula for CPU availability is:

$$\overline{c} = \frac{1}{\max(1,\ L)} \quad = \begin{cases} 1, & \text{if } L \ \leq \ 1 \\ 1/L, & \text{if } L \ > \ 1 \end{cases} \tag{3.1}$$

and the lower-bound CPU availability model is:

$$\underline{c} = \frac{1}{1 + (\frac{n-1}{n}) \times L}. \tag{3.2}$$

The upper-bound model represents the best case CPU availability, which is realized by the example of Figure 3.4 in which none of the threads use the CPU resource concurrently. Provided that the sum of the usage factors of the threads is less than unity, then it is possible that the CPU availability could be as high as unity (i.e., 100%). When the sum of the CPU usage factors is greater than unity, then the best possible value for CPU availability is $1/L$. The lower-bound model is associated with a situation in which the threads' usage of the CPU resource has maximum overlap, as depicted in Figure 3.3 in which all threads always use the CPU resource concurrently.

3.5.3 Bounds for Multi-core Machines

For a multi-core machine with r cores, the following models define upper- and lower-bounds for CPU availability. The upper-bound model is:

$$\overline{c} = \frac{1}{\max(1, \frac{L}{r})} = \begin{cases} 1, & \text{if } L/r \leq 1 \\ r/L, & \text{if } L/r > 1 \end{cases} \tag{3.3}$$

and the lower bound is:

$$\underline{c} = \frac{1}{1 + (\frac{n-1}{n}) \times (\frac{L}{r})}. \tag{3.4}$$

Note that Eqs. 3.3 and 3.4 are generalizations of Eqs. 3.1 and 3.2, *i.e.*, for the case $r = 1$, Eqs. 3.3 and 3.4 are identical to Eqs. 3.1 and 3.2.

Figure 3.9 shows plots of the upper and lower bound formulas for the case $r = 4$ and $n = 16$. The difference between upper- and lower-bounds can be significant for moderate values of aggregate loading. The difference for multi-core machine can be as high as 0.446 for 8-threads with aggregate CPU loading of 0.50. In the following section, experimental studies are performed to determine actual measured values of CPU availability in relation to these bounds.

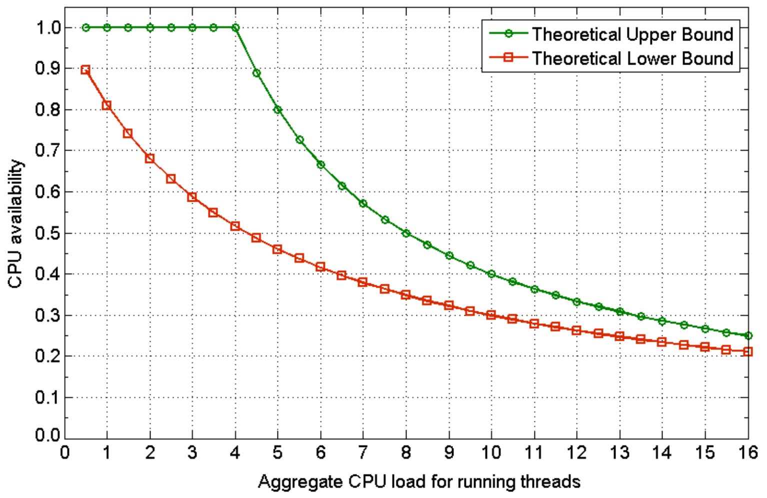


Figure 3.9: Upper- and lower-bounds for CPU availability prediction for $r = 4$ and $n = 16$.

3.6 Empirical Studies

3.6.1 Overview

The purpose of the experimental studies is to empirically measure CPU availability as a function of aggregate loading for collections of threads with randomly selected CPU usage factors. For the study, threads are generated synthetically so that their CPU usage factors can be set accurately. The measured CPU availability associated with a collection of threads executing on a processor is defined by the ratio between the ideal time required to execute one of the threads on an unloaded processor divided by that thread's execution time on a loaded processor. About $2,000$ randomly selected collections of threads are generated for each study; each randomly generated collection of threads provides one measurement of CPU availability. Independent test cases are conducted using this compute-intensive benchmark program to satisfy all possible situations of the CPU model. The major part of the experimental system's flow control is shown in Figure 3.10.

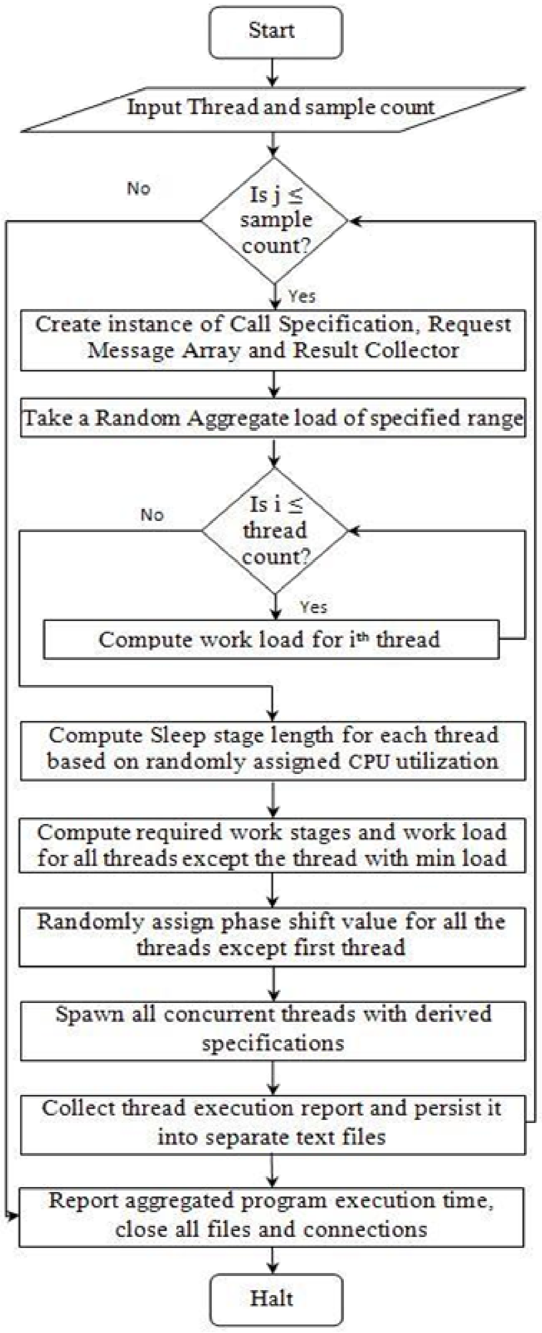

Start
Input Thread and sample count
No
Is j ≤ sample count?
Yes
Create instance of Call Specification, Request Message Array and Result Collector
Take a Random Aggregate load of specified range
No
Is i ≤ thread count?
Yes
Compute work load for ith thread
Compute Sleep stage length for each thread based on randomly assigned CPU utilization
Compute required work stages and work load for all threads except the thread with min load
Randomly assign phase shift value for all the threads except first thread
Spawn all concurrent threads with derived specifications
Collect thread execution report and persist it into separate text files
Report aggregated program execution time, close all files and connections
Halt

Figure 3.10: Flow chart for load distribution and execution process of threads.

Algorithm 1 presents major parts of the experimental system. The algorithm takes the number of concurrent threads and test runs as input and outputs the measured execution efficiency for the batch. Uniform sampling of data across the values of possible aggregate CPU loading has been ensured. A test run of each batch of threads provide one measurement (the minimum efficiency is considered for accuracy) of CPU availability.

Algorithm 1 Aggregate load distribution and measurement of execution efficiency for benchmark threads.

```
Input: Number of threads ($n$), and number of test runs ($\Gamma$)
for count ← 1 ... $\Gamma$ do
    Select a random aggregate CPU load, $L \in [(\epsilon \times n) ... n]$
    for i ← 1 ... $(n-1)$ do
        $\alpha_l = Max\left(L - \sum_{j=1}^{i-1} T_j - (n-i)), \epsilon\right)$
        $\alpha_u = Min\left(L - \sum_{j=1}^{i-1} T_j - (n-i) \times \epsilon), 1.0\right)$
        Select $T_i$ randomly so that $T_i \in [\alpha_l, \alpha_u]$
        $T_i \leftarrow (\alpha_u - \alpha_l) \times T_i + \alpha_l$
    end for
    $T_n \leftarrow L - \sum_{i=1}^{n-1} T_i$
    Compute sleep phase length for each thread using (Eq. 3.5)
    Compute total work amount for each thread depending on CPU loading value
      using (Eq. 3.6)
    Create pipes to communicate between parent and child threads
    Spawn benchmark threads concurrently into single- and multi-core systems
    Wait for threads to complete assigned work and terminates
    Take end time snapshots for each thread after they terminates
    Collect and Persist thread execution data in respective CSV files
end for
Close all files, pipes, and connections
Output: Measured thread execution data
```

To ensure a uniform sampling of data across the values of possible aggregate loadings, a random value of aggregate loading between $(\epsilon \times n)$ and n is chosen first. The value of $\epsilon = 0.05$, denoting a 5% CPU-load, is used to represent the extreme lower CPU load value for a thread. A thread can not have CPU load value of 0.0, else it would never complete its assigned work. The selected aggregate load (*sum-total*) is then distributed among threads using expressions inside the inner for loop. For example, if a thread batch contains 8-threads then the min-limit is 0.4 (0.05×8) and the max-limit is 8.0 (i.e., all CPU-bound threads). Then a random CPU load value between 0.4 and 8.0 is chosen and distributed among 8 threads using Algorithm 1.

The expressions of α_u and α_l is introduced to provide an upper- and lower-limit of available CPU load for the i^{th} thread. A random CPU load value is selected from the range (α_l, α_u) and assigned to the i^{th} thread. The CPU load for the thread is then scaled and placed into T_i. For example, for a scenario having two threads, a value of aggregate loading is chosen between a small value ($\epsilon \times 2$) and 2.0; denote this value as L. Then a random value is chosen between $\max\{\epsilon,\ (L - 1.0)\}$ and $\min\{1.0,\ L\}$, which defines the CPU usage factor of the first thread, say T_1; the CPU usage factor of the second thread is then defined as $T_1 = L - T_2$. In general, for n threads, Algorithm 1 is used to randomly assign the CPU usage factors for a given value of aggregate loading L.

Based on CPU load of each thread, total amount of work (upper range) and sleep phase length values are derived. During each work phase, threads accomplish a fixed amount of work. Threads need to run several phases to complete the total amount of work. As described in Algorithm 1, a phase shift measured value is also assigned to each thread ranging from 0 to length of the phase which provides a degree of staggering of phases among the threads. Figure 3.4 is an example of the best-case staggering; whereas Figure 3.3 is an example of worst-case staggering in terms of CPU availability.

3.6.2 Empirical Environment

The system used for evaluating the *single core* test cases is an Intel Xeon CPU E5540 with 2.53 GHz clock speed, $1,333$ MHz bus speed and 4 GB of RAM. The system used for evaluating the *multi-core* test cases is an Intel Core 2 Quad (quad-core processor), 2.83 GHz clock speed, $1,333$ MHz bus speed with 4 GB of RAM. Due to the different configurations of the single- and quad-core systems, benchmarking for the single-core and quad-core machines were determined separately. The JVM used for these experiments is JDK 1.6. Threads deployed here are independent tasks, meaning there are no interdependencies among threads such as message passing. Threads are spawned concurrently with workloads and phase shifts as described in the previous section. When a collection of threads completes, a report of the threads' execution is produced, which contains *start time, work time, sleep time, number of phases*, and *end time*.

3.6.3 Benchmarking

Benchmarking a thread on an unloaded system enables the calibration of parameters associated with the work and sleep portions of the phases to synthesize a particular CPU usage factor. For setting up the benchmarks, each thread is assigned a *total work* and *sleep phase* duration depending on the assigned CPU load value discussed in Algorithm 1. In the work portion of each phase, threads accomplish a fixed amount of CPU intensive work for generating prime numbers (based on the benchmark program). Depending on the work phase completion time (which is mostly constant for a machine) and CPU load, the sleep phase length of the thread is calculated using the following equation:

$$\mathit{Sleep} = \left(\frac{WorkPhaseTime \times (1.0 - CPUload)}{CPUload} \right) \tag{3.5}$$

According to Eq. 3.5, the sleep time increases as the CPU usage decreases. Thus, a thread with low CPU usage will sleep longer than other threads having higher CPU usage factors. Though the work portion length of threads are same, due to sleep phase length variation due to CPU load, the total phase length (work+sleep) is different for threads.

For setting up the benchmarks, the thread with minimum CPU requirement is assigned 2.0×10^8 units of synthetic CPU work. The thread with minimum CPU load needs to accomplish this total work in 50 work-sleep phases. In the work component of each phase, this thread accomplish 4.0×10^6 units of work. In a time frame, say 10 sec., the amount of work a thread with higher CPU load can accomplish is more than the amount of work a thread with lower CPU load. Thus, it is needed to calculate the variable amount of work required to accomplish by threads to finish assigned work and terminate at the same time. Thus, the total amount of work that needs to be accomplished by other threads can be defined as follows:

$$Work\ load = \left(\frac{MaxPhaseTime}{PhaseTime\ of\ T_i} \times (StageCount \times StageWork)\right) \tag{3.6}$$

Parameters of Eq. 3.6 are the maximum phase time (work + sleep phase time) among all threads, phase time of the thread, number of stages to complete the total ideal work, and amount of accomplished work in each phase. Measured total workloads are then assigned to respective benchmark threads so that the batch terminates at the same time for accuracy.

3.6.4 Empirical CPU Availability Case Studies

3.6.4.1 Case Studies for Single-core Machine

For measuring the CPU availability of the single-core processor, three case studies were conducted in which multiple (2, 3, and 4) threads were spawned concurrently. An aggregate CPU load L was selected randomly, and distributed among the threads as described in Section 3.6.1.

Figures 3.11 and 3.12 show measured CPU availability scatter graphs for 2 and 4 concurrent threads executing on the single-core processor, superimposed with the plots of the upper and lower bound formulas derived in Section 3.5. In these figures, the horizontal axis represents aggregated CPU load and the vertical axis represents CPU availability. Each small dot in these graphs is an independent test case measurement of CPU availability. There are $2,000$ dots in each figure representing

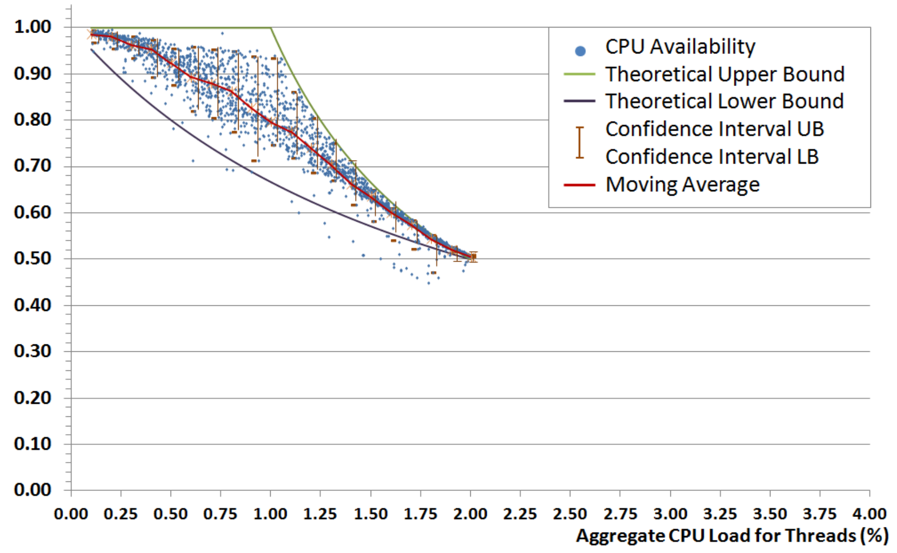


Figure 3.11: CPU availability of 2 *threads in a single-core* machine. Results of 2,000 test cases with upper- and lower-bounds and 90% confidence interval bars.

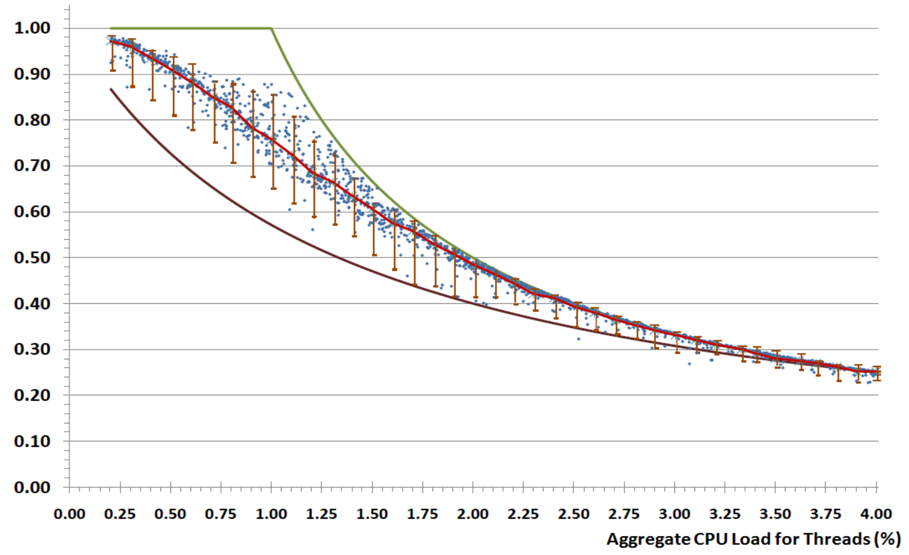


Figure 3.12: CPU availability of 4 *threads in a single-core* machine. Results of 2,000 independent test cases, bounds, and 90% confidence interval bars.

measured CPU availability value among the concurrent threads. A moving average line is also drawn through the data on the graphs for helping to visualize the average measured performance. A window size of 0.10 aggregate CPU load and incremental value of 0.01 was used to calculate the moving average values. A similar sliding window approach was employed to calculate the 90% confidence interval upper and lower limits.

It is apparent from Figures 3.11 and 3.12 that the variation in CPU availability is low when the aggregate CPU loading is either relatively low or relatively high. Thus, CPU availability prediction is quite accurate when the CPU is either lightly or heavily loaded. When the total CPU loading is moderate, the measured CPU availability has more variation and thus it is less predictable. One of the intuitive reasons for low variation when threads have small CPU loading factors is their sleep phase lengths are wider, which decreases the probability of work portion overlap among the threads. On the other hand, when the aggregate CPU load is large, threads have smaller sleep phase and longer work phase lengths which almost always forces work phase to overlap and decrease performance, but in a predictable way.

In further reporting the results of the studies, it is convenient to define the normalized aggregate load, L/n, which is the aggregate load L normalized by the number of threads n. For sample values of normalized aggregate load, Table 3.1 shows the average measured CPU availability (Avg.), the difference in the upper- and lower-bound models (BD) and the difference in the 90% confidence interval limits (CI Diff) for 2, 3, and 4 concurrent threads on the single-core processor.

Table 3.1 shows that the difference between the upper- and lower-bounds can reach as high as 0.375, for 4 threads and a normalized aggregate loading of 0.20. However, the measured 90% confidence interval difference for the same case is much smaller, around 0.104. The difference of the formula-based bound is more precise when the CPU is lightly or heavily loaded.

Table 3.1: CPU availability data for 2, 3, and 4 threads in a *single-core machine* consists of average measured CPU availability (Avg), differences of bound models (BD), and 90% confidence interval limits (CI Diff).

L/n	**2 Thread CPU Availability**			**3 Thread CPU Availability**			**4 Thread CPU Availability**		
	Avg	**BD**	**CI Diff**	**Avg**	**BD**	**CI Diff**	**Avg**	**BD**	**CI Diff**
0.05	0.985	0.048	0.019	0.973	0.094	0.023	0.971	0.133	0.050
0.10	0.980	0.091	0.033	0.946	0.167	0.051	0.935	0.232	0.042
0.20	0.953	0.167	0.076	0.873	0.287	0.087	0.828	0.375	0.104
0.30	0.892	0.231	0.121	0.790	0.376	0.149	0.685	0.306	0.117
0.40	0.862	0.286	0.168	0.707	0.277	0.128	0.575	0.170	0.070
0.50	0.796	0.333	0.175	0.610	0.167	0.059	0.484	0.099	0.027
0.60	0.739	0.208	0.112	0.524	0.101	0.034	0.412	0.059	0.020
0.70	0.662	0.126	0.067	0.461	0.059	0.024	0.353	0.035	0.017
0.80	0.600	0.069	0.042	0.403	0.032	0.021	0.311	0.018	0.011
0.90	0.543	0.029	0.033	0.363	0.013	0.014	0.275	0.008	0.007
1.00	0.495	0.000	0.004	0.331	0.000	0.005	0.249	0.000	0.008

3.6.4.2 Case Studies for Multi-core Machine

For measuring the CPU availability of the quad-core processor, three case studies were conducted in which multiple $(8, 12,$ and $16)$ threads were spawned concurrently to relatively compare the performance between single- and quad-core processors. A similar approach has been employed to calculate the moving average and 90% confidence interval values. CPU availability graphs for 8 and 16 threads are shown in Figures 3.13 and 3.14 respectively. Figure 3.13 shows a higher variation of CPU availability compared with Figure 3.11, which shows CPU availability of 2 threads in a single-core machine. This decrease in variation may be explained as follows: During the thread execution life cycle, depending on CPU availability, threads might be allocated in different cores for load balancing which could decrease the probability of work phase overlap.

The empirical results for the quad-core processor also show that the CPU availability prediction is quite accurate when the CPU is either lightly or heavily loaded. When the total CPU loading is moderate, the measured CPU availability has more

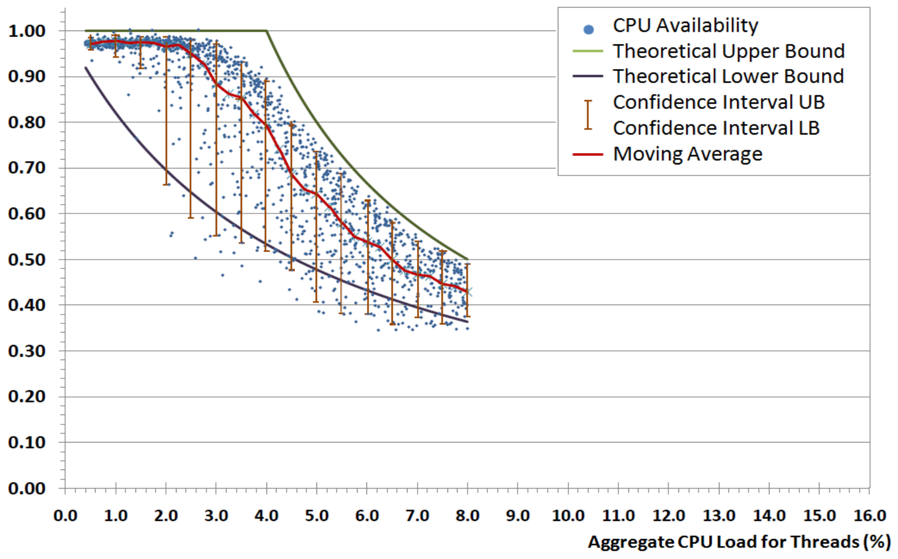


Figure 3.13: CPU availability of 8 *threads in a quad-core* machine. Results for 2,000 independent test cases and 90% confidence intervals.

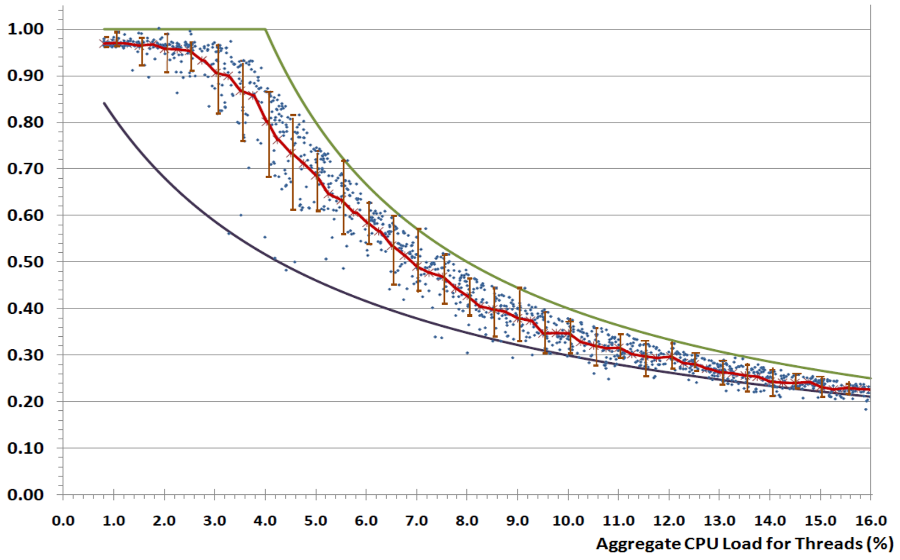


Figure 3.14: CPU availability of 16 *threads in a quad-core* machine. Results of 2,000 independent test cases and 90% confidence intervals.

Table 3.2: CPU availability data for 8, 12, and 16 threads in a *quad-core machine (running Windows)* consists of average measured CPU availability (Avg), differences of bound models (BD), and 90% confidence interval limits (CI Diff).

L/n	**8 Thread CPU Availability**			**12 Thread CPU Availability**			**16 Thread CPU Availability**		
	Avg	**BD**	**CI Diff**	**Avg**	**BD**	**CI Diff**	**Avg**	**BD**	**CI Diff**
0.05	0.973	0.080	0.015	0.971	0.121	0.017	0.969	0.158	0.013
0.10	0.971	0.150	0.019	0.974	0.216	0.022	0.948	0.273	0.058
0.20	0.973	0.260	0.064	0.959	0.355	0.064	0.862	0.429	0.128
0.30	0.956	0.345	0.153	0.904	0.452	0.121	0.690	0.363	0.149
0.40	0.903	0.413	0.322	0.752	0.357	0.085	0.543	0.225	0.116
0.50	0.768	0.466	0.304	0.604	0.246	0.102	0.411	0.152	0.107
0.60	0.595	0.345	0.299	0.490	0.178	0.081	0.344	0.109	0.079
0.70	0.554	0.264	0.222	0.429	0.134	0.073	0.321	0.081	0.057
0.80	0.493	0.208	0.217	0.358	0.104	0.069	0.268	0.062	0.028
0.90	0.446	0.167	0.146	0.328	0.083	0.079	0.240	0.049	0.029
1.00	0.419	0.136	0.114	0.291	0.067	0.058	0.222	0.040	0.023

variation and thus it is less predictable. The empirical results of the quad-core processor also shows that when the numbers of concurrent threads are exact multiples of the number of cores, it provides much better performance than when they are not. That measured variation can be as large as 10% of the overall CPU availability. From the empirical results of single- and multi-core processors, and measured CPU availability scatter graphs, it is apparent that theoretically derived upper and lower limits introduced in this chapter bound actual measured values of CPU availability reasonably well.

For a quad core processor, Table 3.2 shows that the difference between the upper and lower limits formula bounds can reach as high as 0.429, for 16 threads and a normalized aggregate loading of 0.20. However, the measured CPU availability is for this same case is much smaller, around 0.128. The difference of the formula-based bound is more precise when the CPU is lightly or heavily loaded for quad-core processor as well.

3.7 Summary

This chapter has developed analytical models for predicting (and measuring) CPU availability of JVMs supported by single- and multi-core architectures. An extensive set of empirical studies have been conducted in single- and multi-core architecture for verifying the accuracy of the CPU availability prediction models under *Windows environment using Java language.* Thread availability scatter plots provide clear visualization of measured performance based on the density of the dots in the plots. These empirically measured availability values are showed to generally fall within theoretically derived upper- and lower-bound models. A 90% confidence interval for measured availability is shown to provide tighter upper- and lower-limits than the theoretically derived models for upper- and lower-bounds. Additionally, the prediction model for upper- and lower-bound is consistent with the dynamic nature of system and its workload, which can vary drastically in a short amount of time.

As would be expected, degradation in CPU availability occurs when total CPU loading is greater than the total capacity of all CPU cores. In addition to total CPU loading, the total number of concurrent threads is a factor in predicting CPU efficiency; more threads generally incur more context switching overhead, which results in degraded availability. When the total load is less than the total capacity of all cores, the relative alignment of the working and sleeping phases of the threads can have a significant impact on CPU availability. Specifically, increased overlap of the work phases implies lower availability.

It was demonstrated that shifting the relative phasing of the threads to reduce possible work phase overlap can improve the performance (i.e., CPU efficiency). Random aggregate load and phase shift values for concurrent threads were assigned for each for an extensive number of experimental measurements. A thread availability scatter plot provides a clear visualization of measured performance based on the density of the dots in the plot. These empirically measured availability values are showed to generally fall within theoretically derived upper and lower bound formulas. A 90% confidence interval for measured availability is shown to provide significantly tighter upper and lower limits than the theoretically derived formulas for upper and lower bounds.

Chapter 4

CPU Availability Model in Linux Environment

Linux is one of the productive and most popular operating systems in the world. Its acceptance is product of the high and potential capacity that Linux offers to many different fields of work. Linux provides the freedom to run your program (for any purpose), study how the program works, and adapt it to your needs [3]. Linux can be used for high performance server applications, desktop applications, and embedded systems. Because of these positive factors, the CPU availability model is tested under the popular and growing Linux environment to verify the accuracy and usefulness.

There are *three* major contributions of this chapter which are:

- CPU availability model's prediction accuracy under Linux environment.
- Introduction of a compact lower-bound CPU availability model.
- Introduction of an average CPU availability model.

4.1 Introduction

In a multi-processor system, the most widely used approach known as symmetric multi-processing (SMP), where each processor is self scheduling. Each processor has its own private ready queue of ready processes. The scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute. Virtually all modern operating systems support SMP, including Windows 8, Solaris, Linux, and Mac OS X. On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor.

In Linux scheduler 2.6 and newer, all processes have a default static priority. Each CPU has a run-queue made up of 140 priority lists that are serviced in First In First Out (FIFO) order. Tasks that are scheduled to execute are added to the end of their respective run-queue's priority list. Each task has a time slice that determines how much time it's permitted to execute. The first 100 priority lists of the run-queue are reserved for real-time tasks, and the last 40 are used for user tasks [36].

4.2 Empirical Studies in Linux System

4.2.1 Overview

Depending on the process variation (CPU versus I/O bound), the Linux scheduler 2.6 and above dynamically modifies this measured value to assign priority penalty to CPU-bound processes and boost I/O-bound processes. This Dynamic Priority Scheme is controlled with the sleep_avg variable for each process. When an I/O-bound process is awakened from a sleep interval, its total sleep time is added to this variable. In addition, when a process leaves the processor, the time it has been executing on it is subtracted from this variable. The higher the sleep_avg value is, the higher the dynamic priority will be. In addition, the time slice of a process is computed with its priority value, always maintaining its length between the minimum and maximum values [16].

4.2.2 Empirical Environment

The system used for evaluating the multi-core test cases is equipped with Intel(R) Xeon(R) Quad-core W3520 processor, 2.67GHz clock speed, $1,333$ MHz bus speed, and 6.0 GB of RAM. This machine also has Linux kernel version $3.2.0 - 36$. The average CPU load (represents the average system load over a period of time) was 0.0161302 per core in a scale of 1.0 (in a fifteen minute period) before running test cases which indicates that the machines were lightly loaded (essentially unloaded). The *C programming language* in Linux environment is used to implement the analytical model framework and benchmark programs. The *gcc compiler* version used is 4.6.3. Threads deployed here are independent tasks, meaning there are no interdependencies among threads such as message passing. Threads are spawned concurrently in a single- and multi-core machine with measured work load. When the batch of threads complete assigned work and terminates, an execution report is produced, which contains start time, work time, idle time, end time, number of phases, and others for statistical analysis.

4.2.3 Benchmarking

The benchmark programs used for conducting empirical study in Linux environment consists of real-world CPU-intensive programs like *Super prime number generator, Monte-Carlo estimation of* π along with a similar *synthetic benchmark program* (the program used in Chapter 3 as benchmark program but implemented using C language instead of Java) used for verifying the CPU availability model. Table 4.1 consists of a complete list of programs used as benchmark program for this Chapter. The Algorithm 1 presented in Chapter 3 is used to distribute the aggregate load and measurement of execution efficiency for benchmark threads.

Table 4.1: Benchmark programs used for validating introduced CPU, Memory, and Composite prediction models.

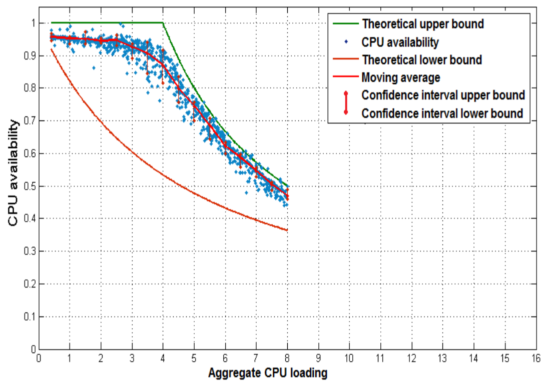
Name	**Description**	**Applied**
piEstimation	Uses Monte Carlo method to estimate the value of π	CPU Availability Model
supPrime	Generates High Order Prime Numbers	CPU Availability Model
linSearch	Linear search program reads from a large data file to search a key	Memory Model
margeSort	Merge sort program to arrange unordered numbers from a large text file	Memory and Composite Model
iRender	Image Rendering of Large BMP files using Image Smoothing Algorithm	Memory Model
smvm	Sparse Matrix Vector Multiplication	Composite Prediction Model
swk	Benchmark program with synthetic work load.	Composite Prediction Model

4.2.4 CPU Availability Case Studies

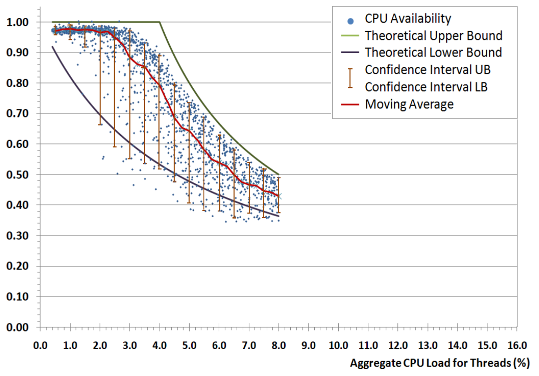
4.2.5 Empirical Results

In this section we have measured the CPU availability of of concurrent threads in Linux environment and compare the outcome with the results obtained in Chapter 3. The obtained results should provide us a clear visualization of efficiency achievement in both paradigms. For measuring the CPU availability of the multi-core processor in Linux, three case studies were conducted in which 8, 12, and 16 threads were spawned concurrently. An aggregate CPU load of L is selected randomly between $(\epsilon \times n)$ and n, and further distributed among n threads as described in Algorithm 1.

Both Figures 4.1 and 4.2 show measured CPU availability scatter graphs for 8 and 16 concurrent threads executing on the quad-core processor, superimposed with the plots of the upper- and lower-bound models derived in Section 3.5.3. The Figure 4.1 (a) show the empirical results under Linux environment using C programming language. On the other hand, the Figure 4.1 (b) show the empirical



(a)

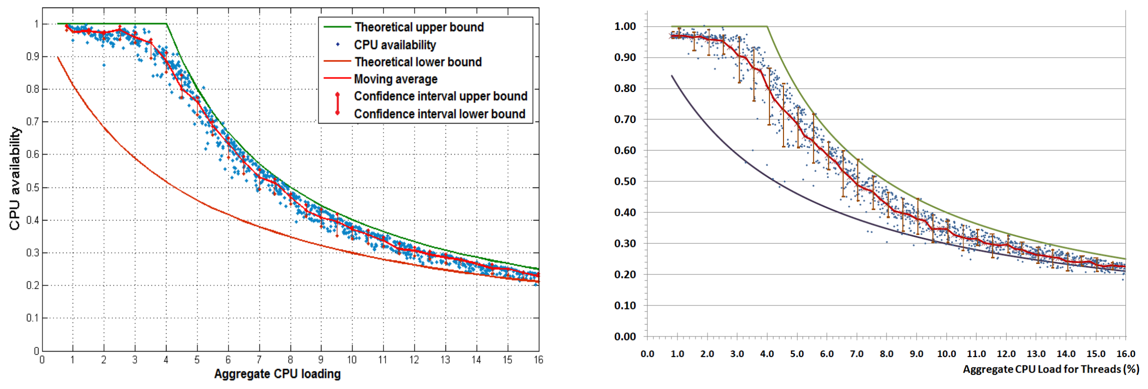


(b)

Figure 4.1: CPU availability of 8 threads in a quad-core machine. (a) Test cases are conducted in Linux systems developed using C language. (b) Test cases are conducted in Windows systems developed using Java language.

results under Windows environment using Java language. Like the test cases of Section 3.6.4.2, each small dot in these figures is an independent test case measurement of CPU availability. There are $2,000$ dots in each figure representing measured CPU availability value among the concurrent threads. A similar moving average line and 90% confidence interval upper- and lower-limits are drawn through the data on the graphs for helping to visualize the average measured performance and confidence limits.

One of the major focus of these empirical work is to verify whether the new empirical results obtained in Linux systems fall within the proposed CPU availability upper- and lower-bounds. It can be seen in both Figures 4.1 (a) and 4.2 (a) that the introduced upper- and lower-bounds can bind the new empirical results very well. Second objective of these empirical study is to compare the empirical results where benchmark programs are developed using C language and executed in Linux environment (shown in Figure 4.1 (a) and 4.2 (a)) with the empirical results of Chapter 3 (shown in Figure 4.1 (b) and 4.2 (b)). A relative comparison between two figures illustrates that the CPU availability dots under C language running in Linux environment has less variation compared with the dots of Java language running in



(a) (b)

Figure 4.2: CPU availability of 16 threads in a quad-core machine. (a) Test cases are conducted in Linux systems developed using C language. (b) Test cases are conducted in Windows systems developed using Java language.

Windows environment. The average CPU availability of Figure 4.1 (a) is better compared with the Figure 4.1 (b) average line. The less variation of dots in Figure 4.1 (a) results in tighter confidence interval bounds compared with the Figure 4.1 (b). The same effect can be observed in Figures 4.2 (a and b) as well. In Linux environment, the measured 90% confidence interval maximum difference is much smaller, around 0.113 compared with 0.304 in Windows environment using JVM. The difference of the formula-based bound is more precise when the CPU is lightly or heavily loaded for quad-core processor as well. Additionally, the empirically-based values for CI Diff (which is tighter than formula based model) can be used as a basis for creating sharper estimates of CPU availability.

The empirical results for the quad-core processor under Linux environment show that the CPU availability prediction is fairly accurate when the CPU is lightly, heavily, or even moderately loaded. When the total CPU loading is moderate, the measured CPU availability has less variation compared with the Windows environment as well. It can be observed from Figure 4.1 (a) and 4.2 (a) that due to efficient thread execution in Linux environment, the gap between average measured CPU availability and lower-bound becomes huge. Thus, it is possible to further tighten the lower-bound so that the bound difference between upper and lower is less and the CPU availability model is more useful (lower prediction range) and attractive even when the CPU loading is moderate. The next section introduces a new lower-bound to *reduce* the bound gap.

4.3 Sharpened Lower-bound Model

The difference between upper- and lower-bounds can be significant for moderate values of aggregate loading. The difference for multi-core machine can be as high as 0.446 for 8-threads with aggregate CPU loading of 0.50. To address this challenge, based on the possibility of having a tighter lower-bound, a sharpened CPU availability lower-bound model for multi-core systems is introduced in this section.

4.3.1 The Model

The sharpened lower-bound model is derived from the lower-bound model introduced in Section 3.5.3. As described earlier, the lower-bound model is based on the worst case staggering of threads which leads to severe performance degradation due to contention among all running threads. But the *probability of worst case staggering* increases when the value of aggregate CPU loading increases. That is, 3 threads in a batch with aggregate CPU load of 0.3 has a lower probability of worst case staggering compared with 3 threads with aggregate CPU load of 0.9.

The new model consists of two scenarios. In the first scenario, the aggregate CPU load value (L) is less than the number of CPU cores (r). Threads are lightly loaded in this scenario and the probability of worst case staggering is small. The following estimated value is subtracted from the denominator of the Eq. 3.4 to provide less efficiency penalty. The additional parameter to this model is the *core hyper-threading*, defined as ξ (cores can have two or more hardware threads).

$$\left(\frac{L}{n \times \xi}\right) \quad \text{if } L \leq r \tag{4.1}$$

In the second scenario, the value of $L > r$. The CPU requirement of the batch increases as the value of L increases. More CPU load increases more contention among running threads and results in degraded performance. As threads' CPU requirement increases, the probability of worst case staggering increases. The following equation is subtracted from the denominator of the eq. 3.4 to provide less penalty based on the value of L.

$$\left(\frac{n - L}{n + r \times \xi}\right) \quad \text{if } L > r \tag{4.2}$$

Based on eq. 4.2, when the value of $L = n$, the worst case staggering possibility is certain and nothing will be subtracted from the denominator. By subtracting the above equations from the introduced lower-bound model, we get the following sharpened lower-bound model.

$$\underline{c} = \begin{cases} \frac{1}{1+\left(\frac{n-1}{n}\right)\times\left(\frac{L}{r}\right)-\left(\frac{L}{n\times\xi}\right)} & \text{if } L \leq r \\ \frac{1}{1+\left(\frac{n-1}{n}\right)\times\left(\frac{L}{r}\right)-\left(\frac{n-L}{n+r\times\xi}\right)} & \text{if } L > r \end{cases} \tag{4.3}$$

Figure 4.3 plots the upper- and both lower-bound models for the case $r = 4$ and $n = 16$. In this figure, the horizontal axis represents aggregate CPU load of the running threads (in a scale of 0 *to* n) and the vertical axis represents CPU availability (in a scale of 0.0 *to* 1.0). In an unloaded system, when the aggregate CPU load for running threads are below or equal to the number of CPU cores, the availability is 1.0 (representing 100%) because every thread gets sufficient CPU resources. When the aggregate CPU load reaches above the number of CPU cores, CPU availability to threads decreases resulting in efficiency degradation. Note that the difference in upper- and lower-bounds becomes significantly tight for moderate values of aggregate loading for the new lower-bound model. In the following section,

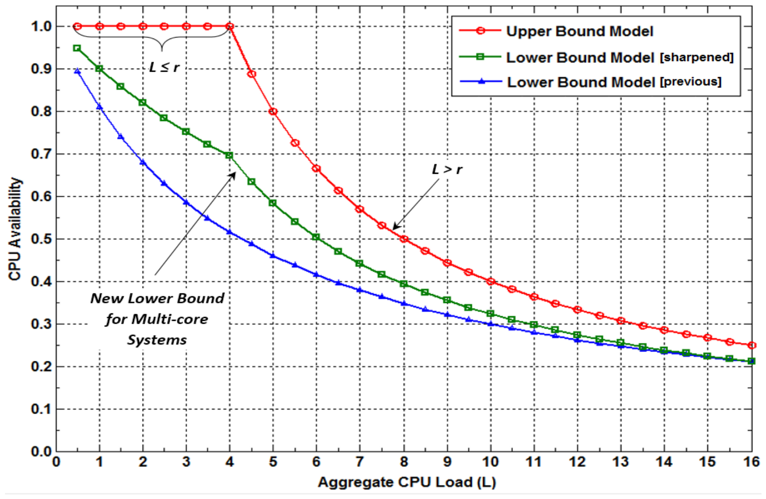


Figure 4.3: Upper- and lower-bounds for CPU availability prediction for $r = 4$ and $n = 16$.

the sparpened lower-bound model is superimposed with old and new empirical data to validate the model.

4.3.2 Sharpened Lower-bound Model Verification

In this section, two sets of empirical data are used to validate the introduced sharpened lower-bound model. For the first set, the empirical test data generated in Section 4.2.5 is superimposed with the sharpened lower-bound model for both 8 and 16 concurrent threads in a batch. For the second set, new data are generated for both 8 and 16 concurrent threads and the sharpened lower-bound model is superimposed on top of the new data. This approach is necessary because our model is based on the observation of the first set of data. The model can be claimed accurate only when it can be validated on new, independent set of data.

CPU availability scatter plots for 8 and 16 threads are shown in Figures 4.4 and and 4.5 respectively utilizing the first set of data. For verifying the accuracy of new lower-bound for multi-core processor, the measured data is superimposed with the upper and *sharpened lower-bound*. Note, the previous lower-bound is replaced with

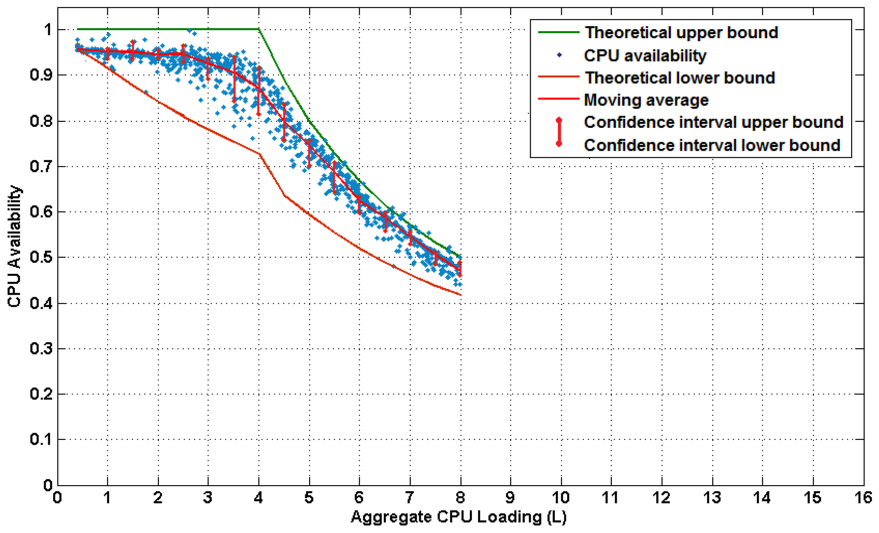


Figure 4.4: CPU availability of 8 threads (for the first set) in a quad-core machine superimposed with tighter lower-bound model. Results for $2,000$ independent test cases and 90% confidence intervals.

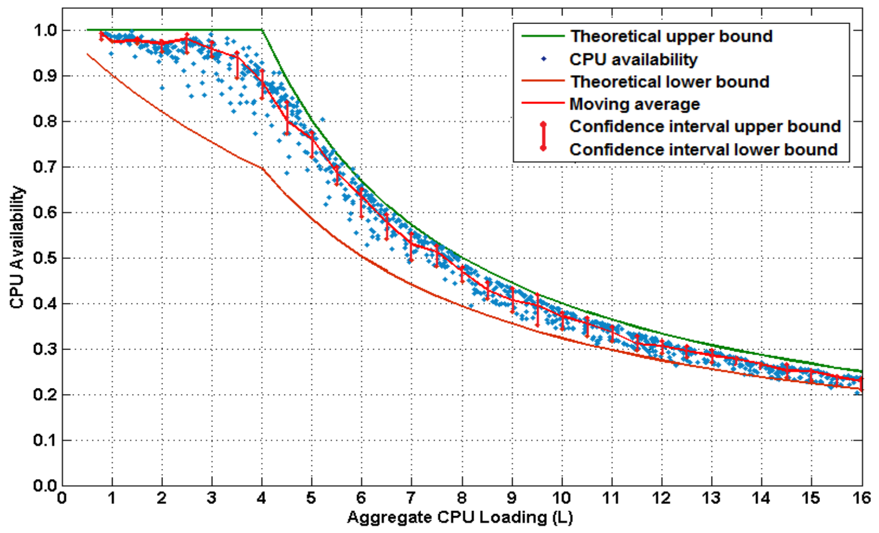


Figure 4.5: CPU availability of 16 threads (for the first set) in a quad-core machine superimposed with tighter lower-bound model. Results for $2,000$ independent test cases and 90% confidence intervals.

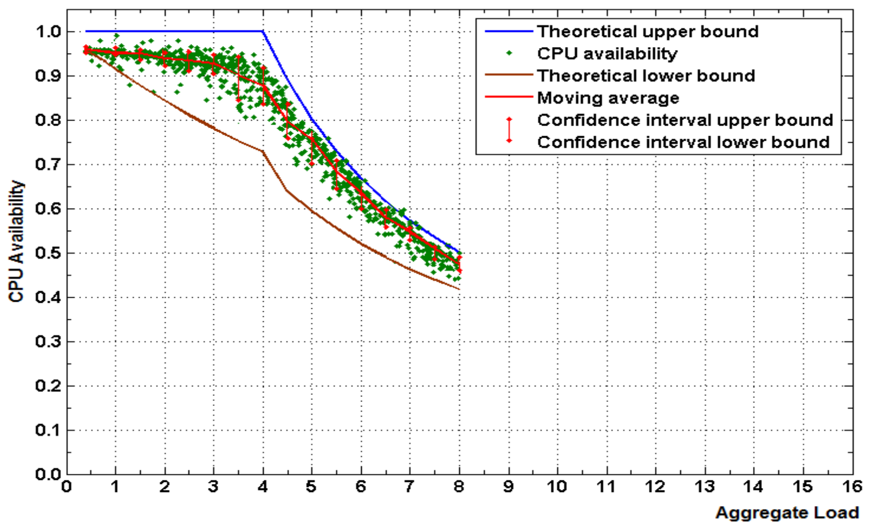


Figure 4.6: CPU availability of 8 threads (for the second set) in a quad-core machine superimposed with tighter lower-bound model. Results for 2,000 independent test cases and 90% confidence intervals.

the new lower-bound for 8, 12, and 16 threads efficiency data. It can be observed from Figures 4.4 and 4.5 that the new lower-bound can bind the efficiency data very well for both 8 and 16 concurrent threads' measured efficiency dots. The introduced lower-bound model reduces the gap between measured average efficiency line and the lower-bound.

For the second set of data, 2000 independent test runs are conducted for both 8 and 16 concurrent threads to validate the model with new set of data for accuracy. A similar approach explained in Section 4.2.5 has been employed to calculate the moving average and 90% confidence interval values.

CPU availability scatter plots for 8 and 16 threads are shown in Figures 4.6 and and 4.7, respectively, utilizing the second set of data. For verifying the accuracy of the new lower-bound for multi-core processor, the new CPU availability data is superimposed with the upper and *sharpened lower-bound*. It can also be observed from Figures 4.4 and 4.5 that the introduced lower-bound model along with the upper-bound model binds the efficiency data very well for both 8 and 16 concurrent

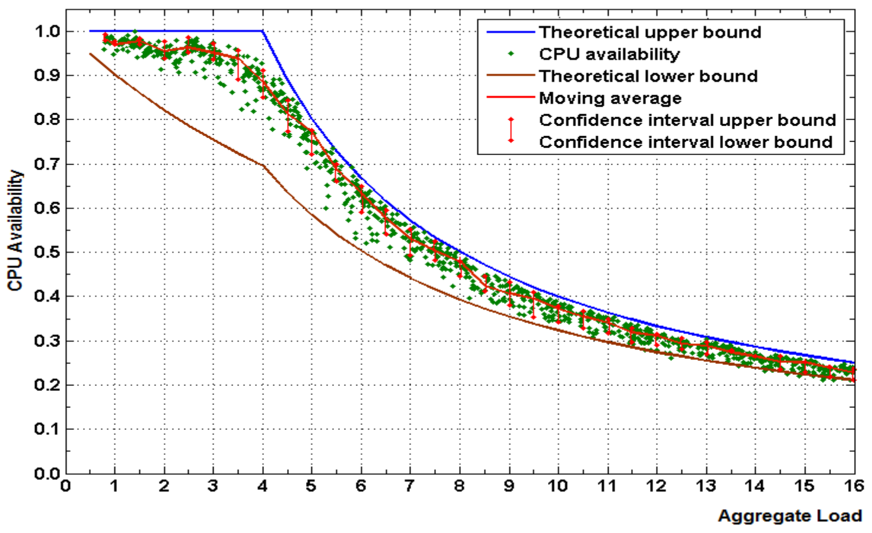


Figure 4.7: CPU availability of 16 threads (for the second set) in a quad-core machine superimposed with tighter lower-bound model. Results for $2,000$ independent test cases and 90% confidence intervals.

threads' measured efficiency dots. Thus, it can be stated with confidence that the sharpened lower-bound model is accurate for multi-core systems.

Table 4.2 shows the average measured CPU availability (Avg.), the difference in the upper- and *new lower-bound* models (BD) and the difference in the 90% confidence interval limits (CI Diff) for 8, 12, and 16 concurrent threads on the quad-core processor. It can be observed from the Table 4.2 that the maximum difference between the upper- and lower-bounds is 0.273, for 8-threads when normalized aggregate loading is 0.50. With the previous lower-bound (Eq. 3.4), the bound difference is 0.466 for the same case. The bound difference is reduced as high as 19.3%. Thus, the introduce tighter lower-bound reduces the bound gap when the CPU is moderately loaded and the availability is uncertain.

Table 4.2: CPU availability data for 8, 12, and 16 threads in a *quad-core machine* (Linux system using C programming language) consists of average CPU availability (Avg), bound differences (BD), and 90% confidence interval limits (CI Diff).

L/n	**8 Thread CPU Availability**			**12 Thread CPU Availability**			**16 Thread CPU Availability**		
	Avg	**BD**	**CI Diff**	**Avg**	**BD**	**CI Diff**	**Avg**	**BD**	**CI Diff**
0.05	0.977	0.036	0.023	0.973	0.059	0.019	0.971	0.080	0.014
0.10	0.972	0.070	0.021	0.963	0.111	0.022	0.963	0.149	0.021
0.20	0.965	0.130	0.046	0.946	0.200	0.054	0.943	0.259	0.046
0.30	0.946	0.184	0.034	0.887	0.272	0.062	0.781	0.230	0.054
0.40	0.917	0.231	0.102	0.762	0.259	0.079	0.587	0.149	0.052
0.50	0.879	0.273	0.113	0.621	0.185	0.096	0.469	0.107	0.047
0.60	0.754	0.224	0.079	0.493	0.141	0.085	0.395	0.081	0.066
0.70	0.681	0.168	0.063	0.433	0.112	0.073	0.326	0.065	0.057
0.80	0.587	0.130	0.056	0.361	0.092	0.054	0.289	0.054	0.026
0.90	0.526	0.103	0.037	0.329	0.078	0.039	0.254	0.046	0.028
1.00	0.471	0.083	0.028	0.301	0.067	0.027	0.228	0.039	0.024

4.4 Average CPU Availability Prediction Model

This section introduces a model for predicting the expected (on average) availability of CPU (instead of calculating the upper- and lower-bounds). As illustrated in previous subsection, exact values of CPU availability are difficult to predict because of dependencies on many factors, including context switching overhead, memory speed, CPU usage requirements of the threads, core hyper-threading, the degree of interleaving of the timing of the CPU requirements of the threads, and the characteristics of the thread scheduler of the underlying operating system. Due to the complex nature of the execution environment, an empirical approach is employed here to estimate expected CPU availability.

4.4.1 The Model

Based on the observed shape and values of measured average CPU availability in Figures 4.4, 4.5, and Table 4.2 of Section 4.3, the mathematical model for average CPU availability is derived. It can be observed from Figures 4.4 and 4.5 that the shape of measured average CPU availability has similarities with the upper bound model (derived in Eq. 3.3).

Because of the similarities with the upper-bound model, the average CPU availability model is also derived based on two scenarios. In the first scenario, when the value of aggregate CPU load (L) is less than or equal to the number of CPU cores (r), that is, running threads has low CPU utilization, the likelihood of work phase contention is minimal. This results in less contending work phases among running threads but still some context switching overhead. The CPU availability upper-bound model is based on the situation in which all work portions of threads are phased out (interleaved) but context switching overhead is ignored. The following model represents the context switching overhead when $L \leq r$. This estimated context switching overhead is subtracted from the first scenario of the upper-bound model to reflect the observed behavior.

$$\left(\frac{n-1}{n+r}\right) \times \left(\frac{L}{(n+r) \times \xi}\right) \quad \text{if } L \ \leq \ r \tag{4.4}$$

In the second scenario, when the value of $L > r$, the best available CPU for running threads is considered as r/L by the upper-bound model. This scenario is based on interleaved work phase of running threads. Though this model provides a good estimation but when the value of L increases, threads become more CPU hungry and the contention is more likely due to relatively wider work portions than idle portions. That is, more CPU load increases more CPU contention and more context switching overhead. The following estimated context switching overhead is subtracted from the second scenario of the upper-bound model to reflect the observed behavior.

$$\left(\frac{1}{1+\left(\frac{n-1}{n}\right) \times \left(\frac{L}{r}\right) \times (r \times \xi + \xi)}\right) \quad \text{if } L \ > \ r. \tag{4.5}$$

Eq. 4.6, provides a model/explanation of what has been observed for thread assignment in processor on the average in Section 4.2.4 (and in [7]). The model of

Eq. 4.6 depends on the aggregate CPU load (sum total) of the set of tasks, number of threads, number of processor cores, and number of hyper-threading in cores, denoted by ξ. For a multi-core machine, the following prediction model estimates the average CPU availability for a set of tasks:

$$c_{avg} = \begin{cases} 1 - \left(\frac{n-1}{n+r}\right) \times \left(\frac{L}{(n+r)\times\xi}\right), & \text{if } L \le r \\ \frac{r}{L} - \left(\frac{1}{1+\left(\frac{n-1}{n}\right)\times\left(\frac{L}{r}\right)\times(r\times\xi+\xi)}\right), & \text{if } L > r. \end{cases} \tag{4.6}$$

The model of Eq. 4.6 is derived from Eqs. 4.4 and 4.5 that depends on whether the aggregate CPU load is less than the number of cores or more than the number of cores. In the first situation, CPU resources are lightly loaded resulting in less context switching overhead and better efficiency. In the second situation, threads are moderate to highly loaded (i.e., aggregate CPU load is more than the number of cores), resulting in more context switching overhead and reduced efficiency. The usage of CPU resource for threads has maximum overlap as depicted in Figure 3.3. In general, thread with more CPU load incurs more contention for resources and context switching overhead. Therefore, an estimated context switching overhead is subtracted from the efficiency value in both cases (i.e., when for $L \le r$ or $L > r$) to best-fit the average efficiency plot.

4.4.2 Average CPU Availability Model Verification

In this section, two sets of test data are used to validate the introduced average CPU availability model. For the first set, the empirical test data generated in Section 4.2.5 is utilized for 8, 12, and 16 concurrent threads. The measured average values are taken from Table 4.2 and superimposed with the model data to verify the accuracy. For the second set, new data are generated for 8, 12, and 16 concurrent threads and the average-bound model is superimposed on top of the new average data. This approach is necessary because our model is based on the observation of the first set of data. The model can be claimed accurate only when it can be validated on a new, independent set of data.

Figure 4.8 (a, b, c) corresponds to the first set of experiment data for 8, 12, and 16 batches of threads in a quad-core machine. In this figure, the horizontal axis represents the normalized aggregate load, L/n, and the vertical axis represents the CPU availability for running threads. The average CPU availability line is plotted from the Table 4.2 containing multi-core test data presented in Section 4.3.2. It can be observed that the CPU availability model lines are smooth and follow similar pattern compared with the average lines obtained empirically. As the number of threads in a batch increases, the shape and pattern of the line of prediction model become almost identical to average measured line.

To validate the accuracy of the model, a new set of data is generated for 8, 12, and 16 concurrent threads. Each empirical study consists of $2,000$ independent test runs. A similar approach explained in Section 4.2.5 is also applied here to measure the running average value from the data set. A window size of 0.10 aggregate CPU load and incremental value of 0.01 was used to calculate the moving average values to plot the line. Figure 4.9 (a, b, c) corresponds to the second set of experiment data for 8, 12, and 16 concurrent threads in a quad-core machine. It can be observed that the introduced model's predicted line and the new average line is again similar and well predicted.

Table 4.3 depicts normalized aggregate load, L/n, average measured CPU availability (Avg.), prediction model value (Model), and prediction error in percentage (Error). Table 4.3 shows that the values obtained from the introduced models are similar to the average measured CPU availability values. For a set of 8-threads, the maximum prediction error is 3.06% when normalized aggregate loading is 0.6. For 12-threads, the maximum prediction error is 3.57% when normalized aggregate loading is 0.3. Finally, for 16-threads, the maximum error is 1.40% when normalized

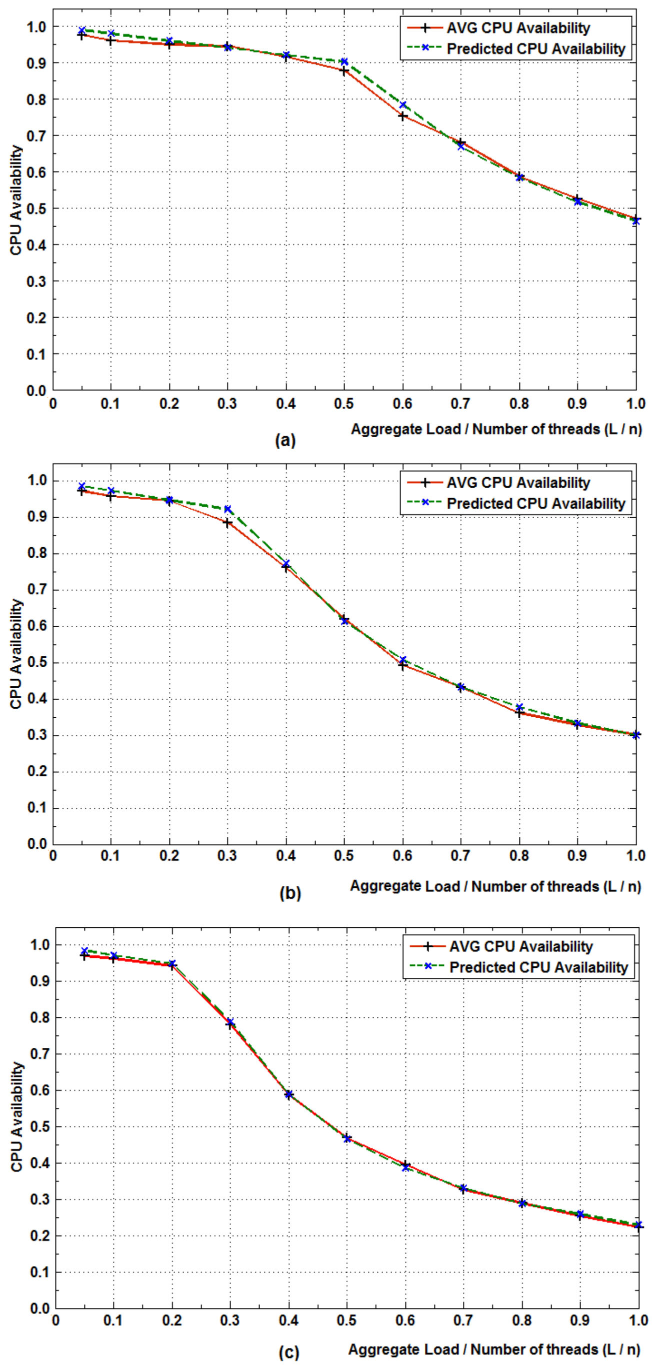


Figure 4.8: The measured average CPU availability data (first set) and predicted average data associated with Eq. 4.6 for (a) Multi-core systems with $r = 4$ and $n = 8$. (b) Multi-core systems with $r = 4$ and $n = 12$. (c) Multi-core systems with $r = 4$ and $n = 16$.

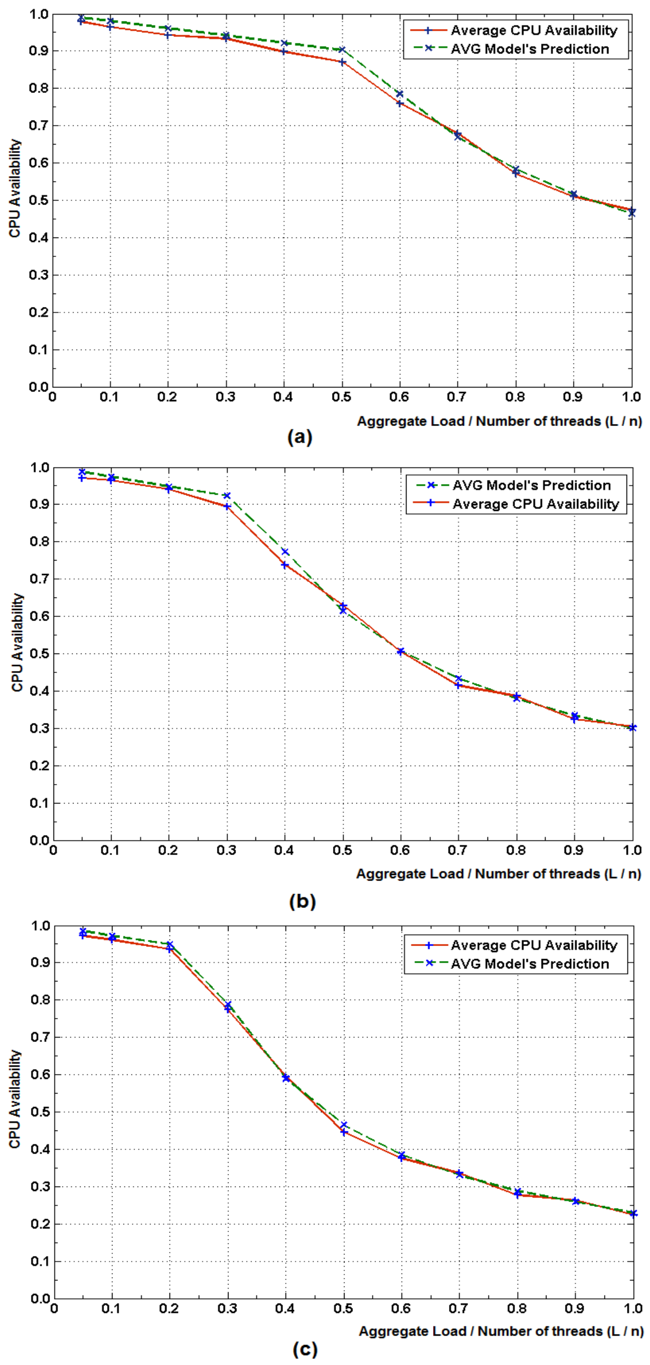


Figure 4.9: The average CPU availability new data (second set) and predicted average data associated with Eq. 4.6 for (a) Multi-core systems with $r = 4$ and $n = 8$. (b) Multi-core systems with $r = 4$ and $n = 12$. (c) Multi-core systems with $r = 4$ and $n = 16$.

Table 4.3: Measured *prediction error for the first set of data* for 8, 12 and 16 threads in a quad-core machine.

L/n	**8 Thread CPU Availability**			**12 Thread CPU Availability**			**16 Thread CPU Availability**		
	Avg	**Model** (c_{avg})	**Error (%)**	**Avg**	**Model** (c_{avg})	**Error (%)**	**Avg**	**Model** (c_{avg})	**Error (%)**
0.05	0.977	0.990	1.33	0.973	0.987	1.41	0.971	0.985	1.40
0.10	0.962	0.981	1.86	0.959	0.974	1.52	0.963	0.970	0.70
0.20	0.951	0.961	1.01	0.946	0.948	0.24	0.943	0.940	0.30
0.30	0.946	0.942	0.43	0.887	0.923	3.57	0.781	0.786	0.53
0.40	0.917	0.922	0.52	0.762	0.774	1.18	0.587	0.585	0.20
0.50	0.879	0.903	2.38	0.621	0.614	0.70	0.469	0.465	0.38
0.60	0.754	0.785	3.06	0.493	0.508	1.54	0.395	0.386	0.91
0.70	0.681	0.669	1.17	0.433	0.433	0.05	0.326	0.330	0.36
0.80	0.587	0.583	0.37	0.361	0.378	1.66	0.289	0.288	0.15
0.90	0.526	0.517	0.93	0.329	0.334	0.54	0.254	0.255	0.09
1.00	0.471	0.464	0.74	0.301	0.300	0.10	0.228	0.229	0.59

aggregate loading is 0.05. The new model has very accurate behavior; the prediction error is always below 4.0 percent.

Table 4.4 contains the new empirical data to validate the model. It can be observed from the table that for a set of 8-threads, the maximum prediction error is 3.24% when normalized aggregate loading is 0.5. For 12-threads, the maximum prediction error is 3.63% when normalized aggregate loading is 0.4. Finally, for 16-threads, the maximum error is 1.92% when normalized aggregate loading is 0.5. This analysis shows that the introduced average CPU availability model is consistent with the second set of data as well. Though the model is created from one set of data, it can accurately predict the average CPU availability for a new set of data as well. The maximum predicted error is only 3.63%. Thus, the model dipicts accuracy and reliability and can be deployed in real-world applications for scheduling.

4.4.3 Comparison With Existing Model

A model similar to the average CPU availability model is the Static Process Assignment Prediction Model (SPAP) introduced by Beltran et al. in [1]. The model they

Table 4.4: Measured *prediction error for the second set of data* for 8, 12 and 16 threads in a quad-core machine.

L/n	**8 Thread CPU Availability**			**12 Thread CPU Availability**			**16 Thread CPU Availability**		
	Avg	**Model** (c_{avg})	**Error** (%)	**Avg**	**Model** (c_{avg})	**Error** (%)	**Avg**	**Model** (c_{avg})	**Error** (%)
0.05	0.979	0.990	1.08	0.9714	0.987	1.56	0.9716	0.985	1.34
0.10	0.965	0.981	1.64	0.9653	0.974	0.87	0.9607	0.970	0.93
0.20	0.943	0.961	1.82	0.9408	0.948	0.72	0.9364	0.940	0.36
0.30	0.933	0.942	0.89	0.8942	0.923	2.88	0.7741	0.786	1.19
0.40	0.898	0.922	2.36	0.7377	0.774	3.63	0.5939	0.585	0.89
0.50	0.871	0.903	3.24	0.6288	0.614	1.48	0.4458	0.465	1.92
0.60	0.760	0.785	2.53	0.5054	0.508	0.26	0.3753	0.386	1.07
0.70	0.679	0.669	0.96	0.4137	0.433	1.93	0.3358	0.330	0.58
0.80	0.571	0.583	1.19	0.3857	0.378	0.77	0.2767	0.288	1.13
0.90	0.510	0.517	0.72	0.3235	0.334	1.05	0.2627	0.255	0.77
1.00	0.473	0.464	0.85	0.3038	0.300	0.38	0.2242	0.229	0.48

have introduced can predict the CPU availability for a batch of threads but only in a single-core machine. Their model fails to provide any estimated value when the batch of threads are spawned in a multi-core machine. Moreover, the model did not consider the effect of core hyper-threading which is common. They have validated their model with only small number of tasks (4) in the run-queue (namely Merge sort, FFT, Network Sniff, and Simulator). Their result shows a maximum prediction error is 3.45% [1].

The average CPU availability model introduced in this section is suitable for multi-core systems with hyper-threading enabled. The introduced model is validated using 8, 12, and 16 threads in a batch. From the Table 4.4, it can be observed that the maximum prediction error is 3.63% for 12 threads running concurrently in a quad-core system. Both models provide high accuracy but our model provides additional advantage because it has the capability to work in multi-core environment (newer systems) and designed (also validated) to work with large sets of tasks in the queue.

4.5 Summary

In this chapter, two prediction models are introduced; (a) given a set of tasks, aggregate CPU load of the set of tasks, and number of CPU cores of a machine, we can predict the *lower-bounds* of CPU availability for the set of tasks; (b) given a set of tasks, aggregate CPU load, number of CPU cores, and hyper-threading of a machine, we can predict the *average CPU availability* for the set of tasks.

A wide range of test cases have been conducted in Linux systems using real-world benchmark programs along with synthetic benchmark program for verifying the accuracy of the CPU availability prediction models. All benchmark programs are implemented using C language. Thread availability scatter plots provide clear visualization of measured performance based on the density of the dots in the plots. These empirically measured availability values are showed to generally fall within theoretically derived upper- and lower-bound models and contains less variation compared with test cases under JVM when the CPU loading is moderate. Because of the less variation of measured efficiency data, a tighter lower-bound is introduced which reduces the gap between upper- and lower-bound as high as 19.3%.

Based on the empirical results for validating the CPU availability model, an average CPU availability model is introduced. The empirical studies performed for validating the average CPU availability model shows that the model values follow the same shape and pattern of the experimentally measured average CPU availability lines. This model is suitable for applications that require a single prediction value instead of upper- and lower bounds for dispatching tasks. Using this reliable information one might be able to determine the order in which tasks should be assigned to the system so that the completion time of all the tasks is minimized.

Chapter 5

Memory and Composite Availability Model

5.1 Memory Performance Model

In addition to CPU availability, the success of approaches for assigning threads (or processes) to multi-core systems relies on the existence of a reasonably accurate memory model for estimating the execution efficiency of tasks due to memory availability. This is because there is a strong relationship between a thread's total execution time and the availability of memory resources used for its execution. *Predicting the execution efficiency of tasks when the available memory is less than required memory is critical in making task assignment and scheduling decisions* [1, 14].

5.1.1 Basics of Memory Performance Model

It is customary to keep several processes running in time-shared systems. For a process to execute, it is necessary for the system to swap in the required portion of code and data of the process into the primary memory. Additional pages are loaded only when they are demanded during program execution. As the degree of parallelism increases among processor, over-allocating memory may result in severe performance degradation. Demand paging saves the I/O necessary to load pages that are never used but increases the probability of memory overrun. If one access out of $1,000$ causes a page fault, the system can be slowed down by a factor as high as 40 because of the page fault service time. The effective access time is directly proportional to the page-fault rate [13, 14].

In this section, an extensive collection of empirical measurements taken from systems with multi-core processors and adequate primary memory to provide a basis for validating proposed memory models for estimating the efficiency of process execution for various memory availability. The proposed models are theoretically derived upper and lower bound models for memory availability. Moving average and theoretically derived bound difference statistics from measured memory availability validate the utility of these theoretical models. Case studies for a multi-core machine involve spawning 1, 2, 3 and 4 concurrent threads with fixed amount of memory requirements by the threads. As the purpose of these empirical studies is to measure the effect of memory availability in thread execution efficiency, the number of threads spawned concurrently are always less than or equal to the number of available cores to avoid context switching (and other relevant CPU related) overhead. Each case study includes a collection of about 2,000 sample executions for statistical analysis.

The rest of the chapter is organized in the following manner. Section 5.1.2 discusses the thread execution model used in this chapter; from this model, theoretical derivations of upper and lower bound models for memory availability are provided. Section 5.1.5 presents the empirical studies including benchmarking, case study measurements for systems with multi-core CPUs and adequate primary memory, and statistical analysis of the results. Section 5.3 summarizes the outcome of empirical studies of this chapter associated with observed execution efficiency due to memory availability during thread execution.

5.1.2 Memory Performance Model Bounds

An analytical framework is developed here based on the availability of primary memory for predicting the efficiency associated with concurrent thread execution in multi-core machines. The primary contribution of this section is the derivation of

Table 5.1: Terms and definitions of the memory model parameters.

Terms	**Definition**
$R_i > 0$	R_i is the required number of memory frames by the process i where $i = (1, 2, ..., m)$.
$R > 0$	R is the total required number of memory frames by all processes ($R = R_1 + R_2 + .. + R_n$).
$M_a \geq 0$	M_a is the total number of free memory frames available in the system.
A_c	A_c is the access time of translation look-aside buffer in the cache memory.
A_p	A_p is the access time of the primary memory.
A_v	A_v is the access time of the virtual memory (backing store).
$\rho > 0$	ρ is the data process time for the thread.
B_l	B_l is the backing store latency.
B_s	B_s is the backing store seek time.
κ	Command queue delay of the system.

upper- and lower-bound formulas for the memory model. Table 5.1 summarizes the notation and definitions of required parameters of memory model bounds.

5.1.3 Assumptions

The following are the assumptions for the Memory performance model:

- A batch of threads are spawned concurrently in a single- or multi-core system.
- The single- and multi-core systems in which threads will be spawned are dedicated, meaning they are not loaded with other threads.
- Memory requirement of each thread is known.
- There are no inter-thread communication or message passing among threads.
- Overhead related to the operating system's real-time process execution is negligible.

The model depends on the above specified assumptions. If these assumptions are not specified, then the model may produce prediction errors or may not be completely viable.

5.1.4 The Model

The memory upper-bound model consists of two scenarios. When M_a is greater than or equal to R and cache memory hit ratio is 100% (all page entries are in translation look-aside buffer), there will be a single look-up in the page table and no additional virtual memory access overhead due to page fault service time. In this ideal case, the efficiency will be 1.0.

In the second scenario, when M_a is less than R; due to shortage of required number of memory frames, pages will be swap out from primary memory to the virtual memory. The operating system swaps out pages of inactive process(es) to free sufficient amount of primary memory for the active process so that it can swap in required program code and data for running. The following equation incorporates the overhead related to the number pages that needs to be swapped out to the virtual memory. The equation also considers the initial virtual memory latency and seek overhead.

$$(A_c + A_v) \times ((\sum_{i=1}^{n} R_i) - M_a) + (B_l + B_s) \tag{5.1}$$

For deriving the upper-bound efficiency model, the ideal thread execution time, denoted by τ (derived in Eq. 5.2), is divided by an expression that represents the execution time, which incorporates the virtual memory access time and backing store overhead for the swapped pages. Here, the ideal thread execution time (τ) represents the best possible execution environment in which threads receive the required amount of primary memory in all situations and the cache memory hit ratio is 100%. That is, for each page, there is only one look-up in page table and a single access in primary memory and data are available in the primary memory.

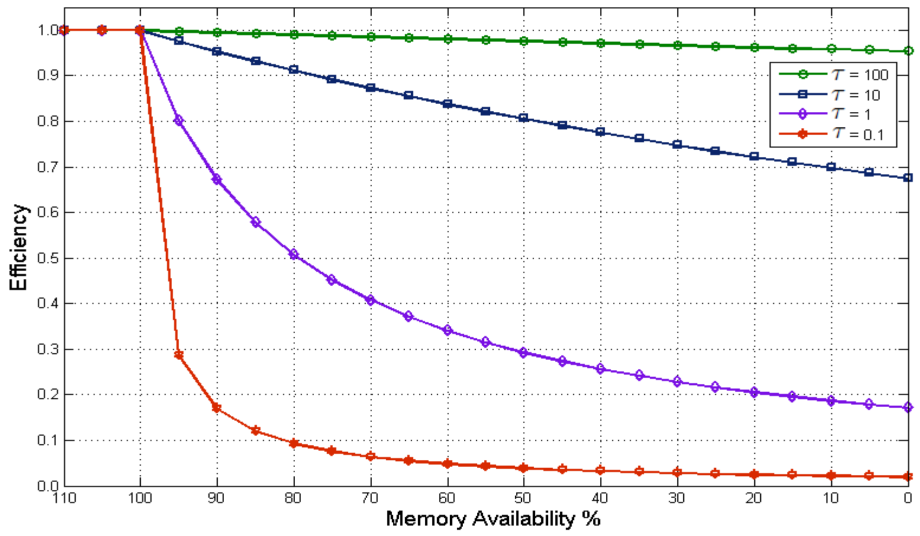


Figure 5.1: Upper-bound efficiency curves associated with Eq. 5.2 for various ideal thread execution time τ in sec.

Thus, the ideal memory access time and the data process time, ρ, represents the ideal thread execution time.

$$\tau = \left\{ (A_p + A_c) \times \sum_{i=1}^{n} R_i \right\} + \rho \qquad \text{if } M_a \geq R \tag{5.2}$$

The upper-bound of the memory performance model, denoted by $\overline{m}$, incorporates both Eqs. 5.1 and 5.2 and can be derived as:

$$\overline{m} = \begin{cases} 1 & \text{if } M_a \geq R, \quad \text{else,} \\ \frac{\tau}{\tau + (A_c + A_v) \times ((\sum_{i=1}^{n} R_i) - M_a) + (B_l + B_s)}. \end{cases} \tag{5.3}$$

Figure 5.1 illustrates a collection of plots for the theoretically derived upper-bound model ($\overline{m}$) for different ideal thread execution times (τ). In this figure, the horizontal axis represents the *memory availability percentage* (the percentage of available primary memory with respect to the required amount of memory by threads) and the vertical axis represents the efficiency of thread execution associated with Eq. 5.3. It can be observed from Figure 5.1 that when the memory availability of the machine decreases, efficiency of the machine for process execution decreases due to increased activity by pager (handling page faults). It can also be observed that when the value of τ is small (i.e., process runs for a short period of time), the efficiency decrease is more dramatic compared with large values of τ. For example, if two processes require the same amount of memory but process one (P_1) runs for a shorter period of time compared to process two (P_2), due to same page fault service time overhead, P_1 has relatively more efficiency degradation compared with P_2, which runs for a longer period of time.

The lower-bound model represents the worst-case execution time and based on two scenarios. In the first scenario, when $M_a \geq R$, and the page entries are not available in the cache memory (cache "miss hit"). In this scenario, there will be an additional lookup of page entries into the primary memory. Thus, the primary memory access overhead is added with the value of τ. The scenario can be expressed by the following equation.

$$\tau + \left(A_p \times \sum_{i=1}^{n} R_i \right) \qquad \text{if } M_a \geq R \tag{5.4}$$

In the second scenario, when $M_a < R$ and page entries are not available in cache memory, this situation additionally includes a primary memory access time because of cache"miss hit", the backing store seek time for individual pages (virtual memory access by multiple concurrent threads will result in storage of pages in non-consecutive order), and the I/O queuing delay. The scenario can be expresses by the following equation.

$$\tau + (A_p \times M_a) + \left\{ (A_v + A_p + A_c) \times ((\sum_{i=1}^{n} R_i) - M_a) \right\} + \eta \tag{5.5}$$

The lower bound model, denoted by $\underline{m}$, additionally includes the backing store overhead expressed in Eq. 5.7 and cache "miss hit" overhead, can be derived by utilizing Eqs. 5.4 and 5.5 as:

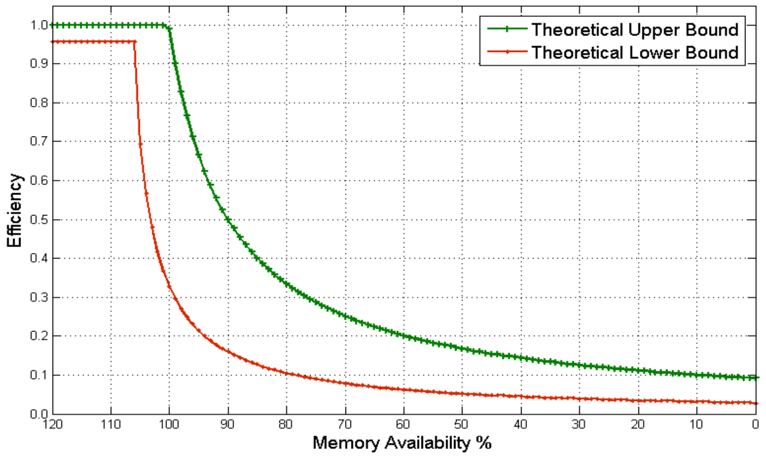


Figure 5.2: Upper and lower bounds for theoretically derived memory availability prediction model for $n = 3$, $r = 4$, and $\tau = 0.5$.

$$\underline{m} = \begin{cases} \frac{\tau}{\tau + A_p \times \sum_{i=1}^{n} R_i}, & \text{if } M_a \geq R, \quad \text{else,} \\ \frac{\tau}{\tau + (A_p \times M_a) + (A_v + A_p + A_c) \times ((\sum_{i=1}^{n} R_i) - M_a) + \eta}. \end{cases} \tag{5.6}$$

and the backing store overhead, denoted by η, considers virtual memory access time for each page and associated command queue delay (κ), can be expressed as:

$$\eta = \left(B_l + \sum_{M_a+1}^{R} (B_s) + \kappa \right). \tag{5.7}$$

Figure 5.2 shows plots of the upper and lower bound models for $n = 3$, $r = 4$, and $\tau = 0.5$ while the memory availability varies from 120% to 0% with respect to the memory requirement by all threads. The backing store overhead (η) increases based on the number of page swap (in/out) and page seek time. The difference in upper- and lower-bound might be significant, when the initial page fault starts, due to the uncertainty of the backing store seek time, latency, and command queue delay, and others. The number of disk commands waiting in the queue is normally the factor that slows down the disk performance by increasing the average disk queue time [7, 44].

Figure 5.2 illustrates the severe efficiency degradation as soon as the pager starts page swap out due to memory shortage. Besides thread, I/O buffers and other associated modules also consume primary memory. For this reason, the page swap out starts a bit early when the memory availability is around 105% (i.e., there is a threshold value after which the pager starts).

5.1.5 Empirical Studies

5.1.5.1 Overview

The purpose of these experimental studies is to measure the efficiency of thread execution as a function of memory availability for collections of threads in an actual dynamic environment. For the study, *two sets of programs* are developed. Algorithm 2 presents the high-level pseudo-code for the first set of programs. The primary role of the first set of programs is to generate threads to allocate and initialize primary memory depending on the memory available percentage and hold the memory until all spawned benchmark threads terminates.

The second set of programs are the memory benchmark programs presented in Table 4.1. The *linear search*, *image rendering*, and *merge sort* programs are used along with a *synthetic benchmark program* for having a precise memory consumption control to cover various scenarios. These benchmark programs are suitable for the empirical studies because they include memory intensive expressions and require reasonable amount of free memory for processing data. Algorithm 3 presents a high-level pseudo-code for creating and spawning the memory benchmark programs.

For the synthetic benchmark program, pages are replaced on demand while the process is running to simulate real-world applications behavior. A page replacement overhead, based on page replacement rate, is integrated in this empirical study. The input parameter of this page replacement overhead model, denoted by φ, includes the memory requirement of i^{th} thread, denoted by tm_i in KB, page replacement rate, denoted by ω, and individual page replacement time. Thus, φ can be defined as:

Algorithm 2 Setting up the memory framework for the benchmark programs by allocating and holding memory.

```
Input: Number of threads ($n$), Number of test runs ($\Gamma$), Memory requirements of each thread ($ts$) in KB, and memory availability percentage ($MAP$).
for count $\leftarrow$ 1 ... $\Gamma$ do
    Select a random memory availability percentage between 0 ... 120
    $R \leftarrow \frac{\sum_{i=1}^{n} ts_i}{page\ size}$, sizes are in KB
    Available free memory frames, $M_a \leftarrow \frac{total\ memory - used\ memory}{page\ size}$ of the system.
    $Consumememory \leftarrow M_a - (R \times MAP) \times 0.01$
    Create child threads to carryout the following tasks:
    for k $\leftarrow$ 1 ... $cm$ do
        Allocate memory, $m = (void\ *)\ malloc(page\ size)$
        Initialize pages, $memset(m,\ 0,\ page\ size)$ to avoid fake memory allocation
        Hold primary memory until all benchmark threads complete execution
    end for
    Generate and spawn threads concurrently for measuring execution efficiency
      (using Algorithm 3).
end for
Output: $R$, $M_a$, and $allocated\ memory$.
```

$$\varphi = \left(\frac{\sum_{i=1}^{n} tm_i}{page\ size} \times \sigma \right) \times \omega. \tag{5.8}$$

Pages are replaced at a rate of ω after completion of each work phase.

In the context of memory management, the Linux kernel creates a new virtual address space for each child thread because threads are created by using the fork() system call. The kernel creates a complete copy of the existing process's virtual address space; then it will copies the parent process's *vm_area_struct* descriptors and will create a new set of page tables for the child [13,14]. The parent's page table and references will be directly copied; thus the parent and child shares the same physical pages in their address space for all test cases.

5.1.5.2 Empirical Environment

The same empirical environment described in Section 4.2.2 is used for the memory studies. As the purpose of these tests is to precisely measure the effect of memory

Algorithm 3 Creating and spawning child threads for measuring thread execution efficiency of memory benchmark programs.

```
Input: Number of threads ($n$), 2D array size ($s$), pipe descriptor ($fd$), and data key.
Wait for the response of consume memory operation
Generate a child process to govern next set of child threads via pipe communication
for i ← 1 ... $n$ do
    Create the child thread $T_i$
    Create communication link (pipe) between parent and child threads
    Take the start time snapshot for the thread $T_i$
    Compute $s_i$ with respect to memory requirement of the benchmark thread
    Allocate memory for the 2D array size of $s_i$ and initialize with random
      numbers $\in [1..10k]$
    Perform measured work based on spawned benchmark program
    Take the end time snapshot for the thread $T_i$ while terminates
    $Execution\,time \leftarrow (start\,time - end\,time)$
    $Efficiency \leftarrow \frac{benchmark\ time}{execution\ time}$
    Transfer $Efficiency, Execution\ time, utilized\ memory\ size$, and
      thread number to its parent via pipe descriptor
end for
Wait for all child threads to terminate and release memory occupied by the service
  thread
Store execution data into respective CSV files for analysis
Output: Complete execution report for the set of threads along with measured
  efficiency.
```

availability on thread execution, the number of concurrent threads *generated using benchmark programs* spawned were always less than or equal to the number of CPU cores to prevent any possible CPU related overhead due to context switching or other CPU-related inefficiencies.

5.1.5.3 Benchmarking

Table 4.1 consists of benchmark programs used for the memory model test cases. Each *linear search*, *image rendering*, and *merge sort* benchmark thread consists of a set of operations that are mix of CPU and memory related expressions that are ideal for the empirical environment of the memory model. For setting up the benchmark, threads are assigned an independent two dimensional array size of 250MB. For an array, memory allocation of rows have been completed first followed by the memory allocation of columns of each row. After memory allocation, each memory frame was initialized with a random value ranging from 1 to $10,000$ to be searched. This initialization also helps to avoid fake memory allocation (due to demand paging). The system function, *meminfo*, is invoked to validate the accuracy of memory allocation for respective threads. For the *iRender* benchmark program, several large bitmap files are rendered using the image smoothing algorithm and the output file is stored in secondary disk. For benchmark runs, available primary memory of the system were above the required memory of threads to ensure 100% availability of primary memory to complete assigned work. This approach ensures no additional overhead due to page fault service. Constants and literals in expressions (for generating work load) are avoided to eliminate caching effects. Machine's efficiency data is stored in multiple CSV files for benchmarking and statistical analysis.

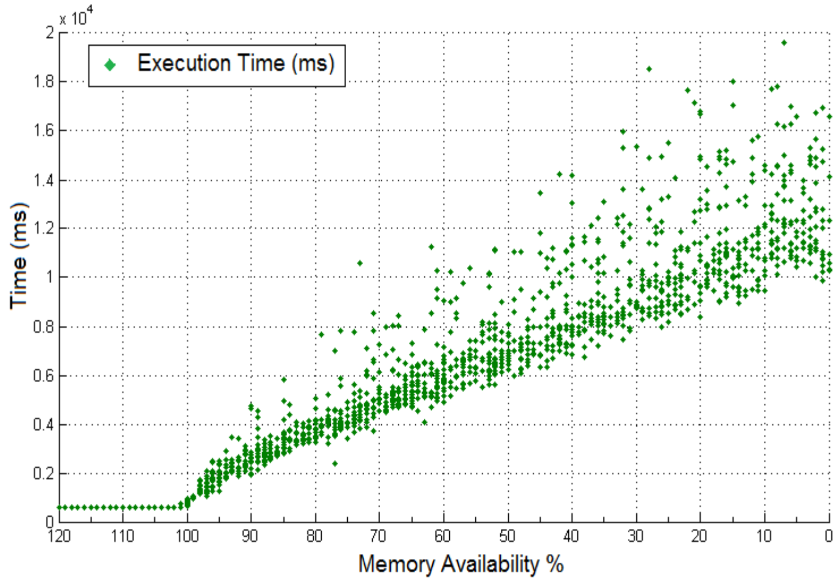


Figure 5.3: Increase in thread execution time while the memory availability was varied from 120% down to 0% for 4 threads.

5.1.6 Empirical Memory Model Case Studies

A wide range of empirical studies were conducted to validate the proposed memory availability model of Eqs. 5.3 and 5.6. For estimating the effect of memory availability in concurrent thread execution, four major independent case studies were conducted in which 1, 2, 3, and 4 threads were spawned concurrently in a quad-core machine. Each case study is incorporated with 2,000 independent test runs for measuring the thread execution time where memory availability was varied from 120% down to 0% with respect to the memory requirement of threads.

Figure 5.3 illustrates the increase in total execution time when the availability of free primary memory reduces. In this figure, the horizontal axis represents the memory availability percentage for the running threads and the vertical axis represents the thread execution time. In this case, 4 concurrent threads are spawned and the available memory of the system is reduced from 120% (with respect to total memory requirements of running threads) down to 0% (no available primary memory). It

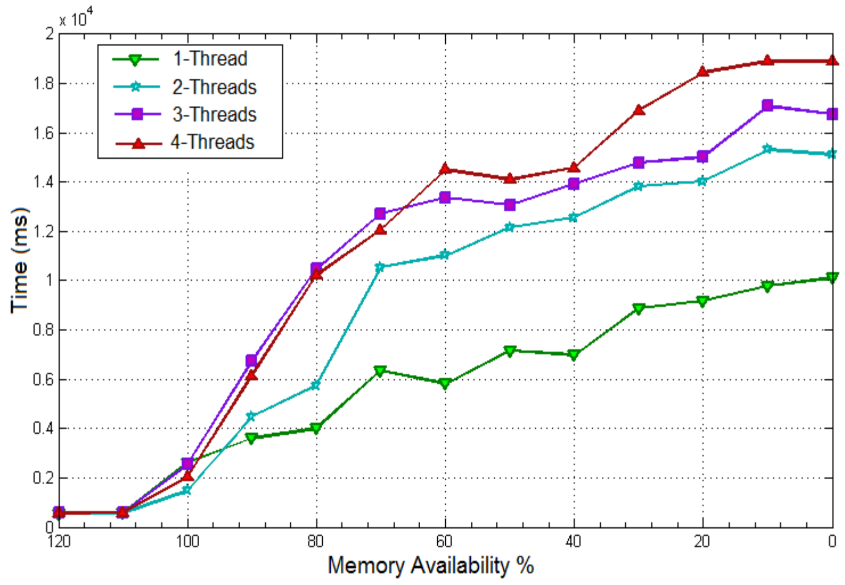


Figure 5.4: A relative comparison of average thread execution time among 1, 2, 3, and 4 concurrent thread(s).

can be observed from the Figure 5.3 that when the available memory falls below 100%, the execution time increases sharply due to page swap-out from the primary memory to backing store.

Figure 5.4 shows measured average thread execution time line graph for 1, 2, 3, and 4 threads spawned concurrently in a quad-core machine. In this figure, the horizontal axis represents the memory availability percentage for the running threads and the vertical axis represents the average thread execution time to complete the assigned job. Average thread execution time were calculated using a sliding window size of 1.0% memory availability and incremental value of 0.1. It is apparent from the figure is that the thread execution times are similar (for thread 1, 2, 3, and 4) when the memory available is 100% or above (in respect to memory requirement of all threads). As the memory availability falls below 100%, the average thread execution time increases sharply for all cases because of the page fault service overhead.

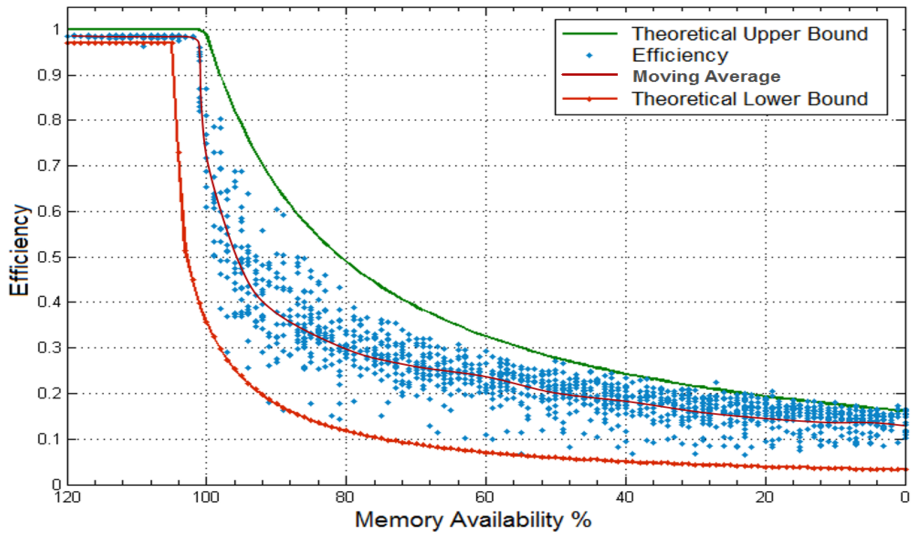


Figure 5.5: Efficiency of thread execution for 1 thread in a quad-core machine. Results of $2,000$ independent test cases along with derived upper- and lower-bounds.

Figures 5.5, 5.6, 5.7 and 5.8 show measured thread execution efficiency scatter graphs for 1, 2, 3, and 4 concurrent threads, superimposed with plots of the theoretically derived upper- and lower-bound models associated with Eqs. 5.3 and 5.6. In these figures, the horizontal axis represents the memory availability percentage (system's memory availability with respect to running threads) and the vertical axis represents efficiency of thread execution (the ideal thread execution time divided by the test run execution time) on a scale of 0.0 to 1.0. Each small dot in these graphs are an independent test case measurement of thread execution efficiency. This empirical study on a quad-core machine shows that as page swap out starts due to lack of primary memory availability, the thread execution efficiency falls sharply because of the page fault service time. The variation in thread execution efficiency is the most, thus less predictable, when the availability is in the range of 100% to 80%. Reasons for wide variation are the initial backing store latency, command queue

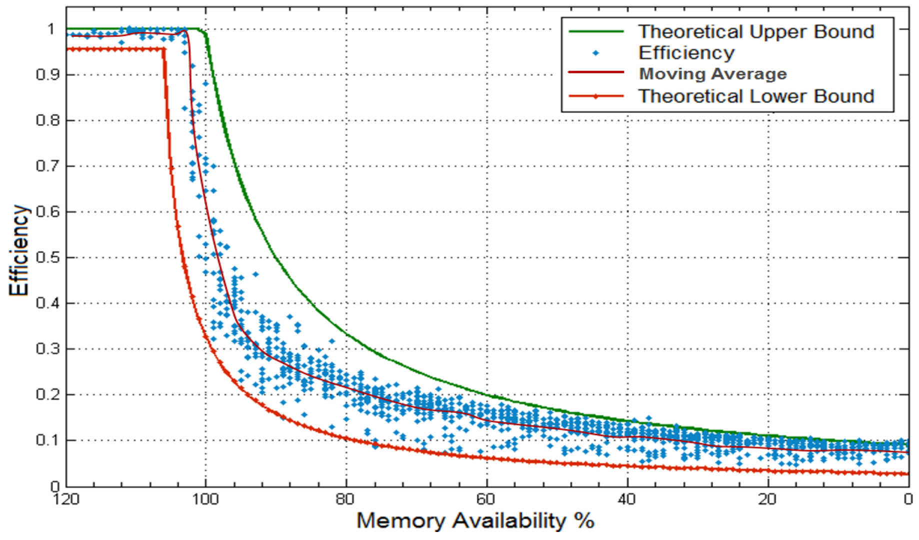


Figure 5.6: Efficiency of thread execution for a set of 2 threads in a quad-core machine. Results of $2,000$ independent test cases along with derived upper- and lower-bounds.

delay, and serial I/O processing. From the figures, it is apparent that the theoretical upper- and lower-bounds introduced in this chapter accurately bind the actual measured values of thread execution efficiency.

Figure 5.8 shows smaller bound difference compared with 1 and 2 thread cases. The variation in execution efficiency reduces when the number of threads in a batch increases. It can also be observed that when the memory availability is above 100%, bound difference increases as the number of thread increases. As the number of thread increases, the probability of page swap out and queuing delay increases. The difference of the model-based bound is more precise when the memory availability for threads is either high or low.

In further reporting the results of the studies of 1, 2, 3, and 4 concurrent threads in a quad-core machine, Table 5.2 and 5.3 presents the value of theoretically derived upper-bound model (UB), lower-bound model (LB), the difference between upper- and lower-bounds (BD), and the average measures thread execution efficiency (AVG) while the memory availability was varied from 0% to 120%. From the table, it

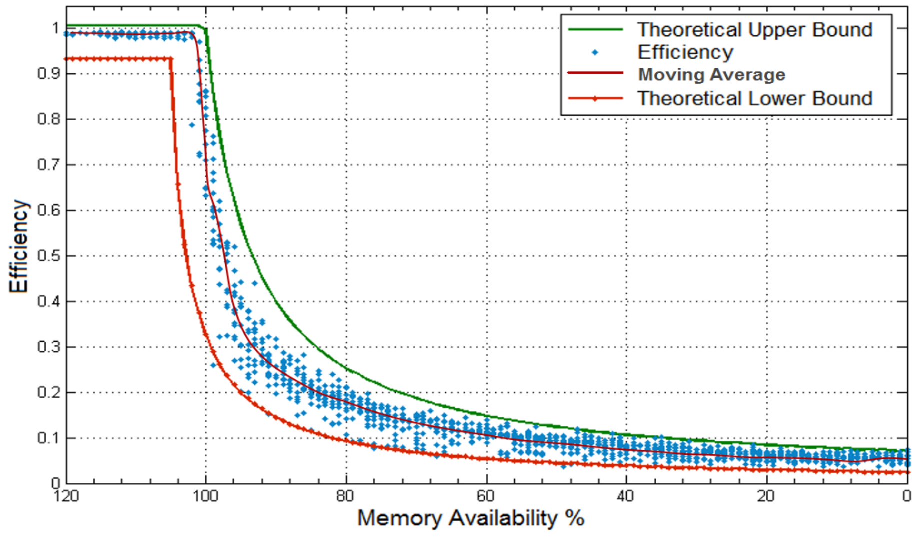


Figure 5.7: Efficiency of thread execution for 3 threads in a quad-core machine. Results of $2,000$ independent test cases along with derived upper and lower bounds.

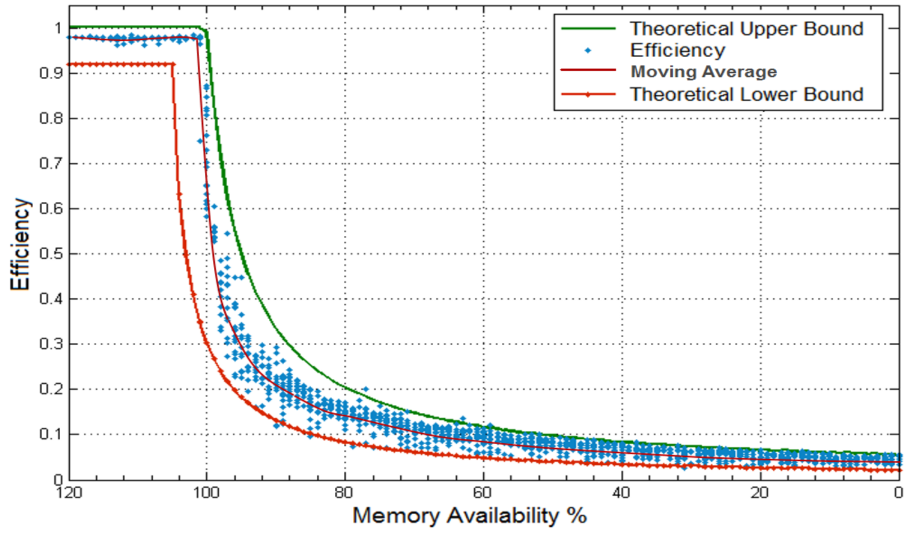


Figure 5.8: Efficiency of thread execution for a set of 4 threads in a quad-core machine. Results of $2,000$ independent test cases along with derived upper- and lower-bounds.

Table 5.2: Average measured thread execution efficiency (AVG), Upper-bound (UB), Lower-bound (LB), and Differences between upper- and lower-bounds (BD) for *1 and 2 concurrent threads.*

MAP	**1 Thread Memory Availability**				**2 Thread Memory Availability**			
	AVG	**UB**	**LB**	**BD**	**AVG**	**UB**	**LB**	**BD**
0%	0.134	0.163	0.030	0.130	0.067	0.092	0.028	0.063
10%	0.139	0.177	0.035	0.142	0.071	0.101	0.031	0.070
20%	0.147	0.177	0.039	0.156	0.080	0.112	0.035	0.077
30%	0.161	0.217	0.044	0.173	0.093	0.126	0.039	0.087
40%	0.179	0.244	0.051	0.194	0.115	0.144	0.045	0.099
50%	0.207	0.279	0.059	0.220	0.131	0.168	0.052	0.116
60%	0.247	0.326	0.071	0.255	0.142	0.201	0.063	0.138
70%	0.261	0.392	0.089	0.303	0.167	0.251	0.079	0.173
80%	0.296	0.491	0.118	0.373	0.218	0.335	0.105	0.229
90%	0.382	0.657	0.178	0.479	0.276	0.500	0.160	0.341
100%	0.439	0.991	0.358	0.633	0.401	0.991	0.328	0.663
110%	1.000	1.000	0.978	0.022	1.000	1.000	0.957	0.042
120%	1.000	1.000	0.978	0.022	1.000	1.000	0.957	0.042

Table 5.3: Average measured thread execution efficiency (AVG), Upper-bound (UB), Lower-bound (LB), and Differences between upper- and lower-bounds (BD) for *3 and 4 concurrent threads.*

MAP	**3 Thread Memory Availability**				**4 Thread Memory Availability**			
	AVG	**UB**	**LB**	**BD**	**AVG**	**UB**	**LB**	**BD**
0%	0.041	0.061	0.024	0.037	0.031	0.046	0.022	0.025
10%	0.044	0.067	0.026	0.041	0.038	0.051	0.024	0.027
20%	0.047	0.075	0.029	0.045	0.041	0.057	0.027	0.031
30%	0.051	0.085	0.033	0.051	0.044	0.065	0.030	0.035
40%	0.061	0.097	0.038	0.059	0.050	0.075	0.034	0.040
50%	0.084	0.115	0.045	0.070	0.061	0.089	0.040	0.048
60%	0.103	0.139	0.054	0.085	0.083	0.108	0.049	0.059
70%	0.139	0.177	0.068	0.109	0.102	0.139	0.062	0.077
80%	0.182	0.244	0.093	0.151	0.146	0.195	0.084	0.111
90%	0.245	0.391	0.145	0.247	0.214	0.326	0.132	0.194
100%	0.381	0.988	0.327	0.662	0.379	0.992	0.303	0.688
110%	1.000	1.000	0.933	0.067	1.000	1.000	0.920	0.081
120%	1.000	1.000	0.933	0.067	1.000	1.000	0.920	0.081

can be observed that the difference between upper- and lower-bounds can reach as high as 0.688 when $n = 4$ and $MAP = 98$ (i.e., when the page fault just starts). However, the measured average thread execution efficiency is for this same case is much smaller, around 0.379. Bound difference are low for the 4 thread case when the memory availability is relatively high or low.

5.2 Composite Prediction Model

The CPU availability and memory models from Chapters 3 and 5, respectively, are used as building blocks in this section to derive a composite CPU/Memory prediction model for estimating the efficiency of thread execution in multi-core systems. As new processes are assigned or existing processes complete execution, the CPU and memory availability of a given machine can change significantly in a short span of time. Existence of the composite prediction model is important because there exists a wide range of applications and scientific models (e.g., geological, meteorological, economical and others) that requires extensive use of both CPU and memory resources, repeatedly. Assigning a set of batch tasks in a distributed environment (distributed schedulers) can utilize the composite model to determine the order (or find subsets) in which the tasks should be assigned to compute nodes so that the completion time for all tasks is minimized prior to its placement in the run queue.

5.2.1 Basics of Composite Prediction Model

As shown in Chapter 4 and Section 5.1, there is a strong relationship between a thread's total execution time and the availability of CPU and memory resources used for execution. It is necessary to have accurate models for estimating CPU and memory resources because of the dynamic nature of computer systems and their workload. As new processes are assigned or existing processes complete execution, the CPU and memory availability of a given machine can change significantly in a short span of time. An extensive collection of empirical measurements are taken from systems with multi-core processors and adequate primary memory to provide a basis for validating proposed analytical model for estimating process execution performance with variable CPU and memory availability.

5.2.2 Bounds of Composite Prediction Model

A composite analytical framework consisting of upper- and lower-bounds are derived for estimating the overall efficiency for a batch of tasks in multi-core systems. Input parameters for the composite upper- and lower-bound models include the CPU availability upper- and lower-bound models derived in Eqs. 3.3 and 4.3, and memory availability upper- and lower-bound models derived in Eqs. 5.3 and 5.6, respectively. The composite CPU and memory availability upper-bound model, denoted by $\overline{cm}$, represents the best case efficiency value of a machine for concurrent thread execution can be derived by the product of two models because CPU and memory are the two primary factors used to characterize machines.

$$\overline{cm} = \begin{cases} \frac{1}{\max(1,\frac{L}{r})} & \text{if } M_a \geq R, \quad \text{else} \\ \frac{1}{\max(1,\frac{L}{r})} \times \frac{\tau}{\tau+(A_c+A_v)\times((\sum_{i=1}^{n} R_i)-M_a)+(B_l+B_s)} \end{cases} \tag{5.9}$$

The values of $\overline{c}$ and $\overline{m}$ represents the relative impact of a machine's overall efficiency due to loading of machine's CPU and memory resources, respectively. The composite model upper-bound model consists of two scenarios. When M_a is greater than or equal to R, the execution efficiency value depends on the value of aggregate CPU loading. In the second scenario, when M_a is less than R, the execution efficiency value depends on the composite effect of CPU and memory.

The composite CPU and memory availability lower bound model, denoted by $\underline{cm}$, represents the worst case efficiency value of the machine for thread execution is defined by the product of CPU and memory lower bound.

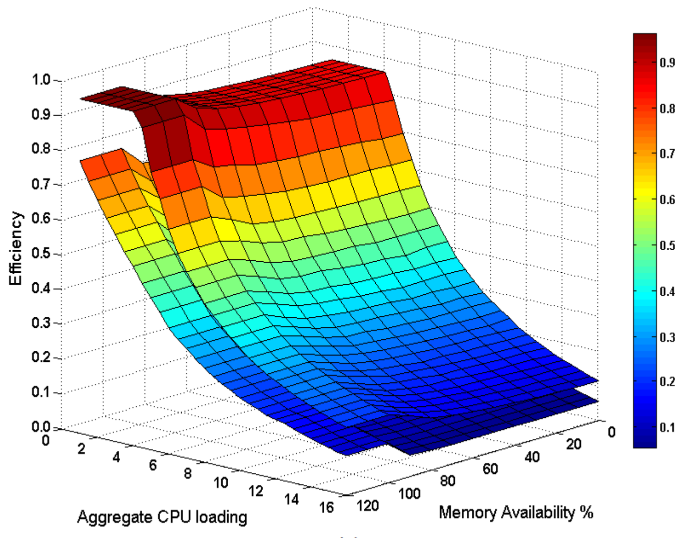
$$\underline{cm} = \begin{cases} \frac{1}{1+(\frac{n-1}{n})\times(\frac{L}{r})-\mu} \times \frac{\tau}{\tau+A_p\times\sum_{i=1}^{n} R_i}, & \text{if } M_a \geq R, \text{else} \\ \frac{1}{1+(\frac{n-1}{n})\times(\frac{L}{r})-\mu} \times \frac{\tau}{\tau+(A_p\times M_a)+(A_v+A_p+A_c)\times((\sum_{i=1}^{n} R_i)-M_a)+\eta} \end{cases} \tag{5.10}$$

A new parameter, μ, is incorporated to represent the probability of worst-case staggering of work-phases. When the value of L is less than or equal to r, the value of μ is $L/(n \times \xi)$ because of the less probability of worst-case staggering. When the value of L is greater than r, the value of μ becomes $L/(n + r \times \xi)$ because of increased work-phase probability.

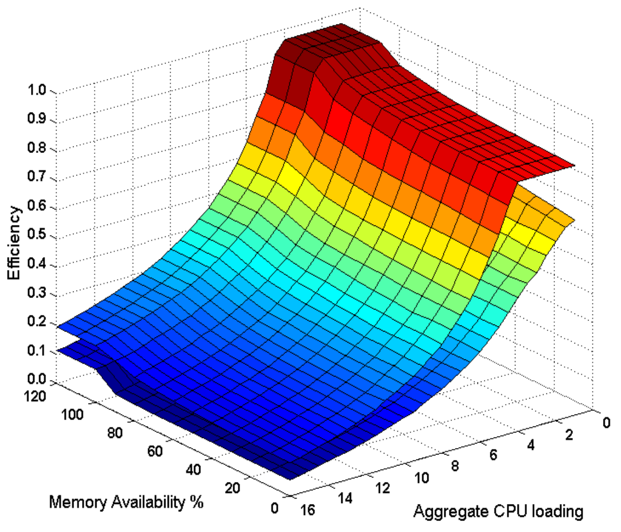
The lower-bound of composite prediction model consists of two scenarios. When the value of M_a is greater than or equal to R, the execution efficiency value is derived from the product of CPU availability lower-bound and memory models' first lower-bound expression. In the second scenario, when M_a is less than R, the execution efficiency value is derived from the product of CPU availability lower-bound model and the memory lower-bound models' second expression to reflect the composite effect accurately.

Figure 5.9 (a, b) shows surfaces of the theoretically derived upper- and lower-bound model in which horizontal axes represents aggregate CPU loading (of the set of tasks) and memory availability percentage (with respect to memory requirements of the set of tasks), and the vertical axis represents the overall efficiency of the machine. It can be observed that for CPU loading values greater than the number of cores (here, $r = 4$), the idealized function for $\overline{c}$ decreases according to the ratio of the number of cores to the total CPU loading. The significant degradation of efficiency value can also be observed when the memory availability for threads decreases accordingly from 120% down to 0%. Ideally, if a machine's CPU and memory resources are both lightly loaded, then the efficiency of the machine is at or near its maximum value.

Figure 5.10 (a, b, and c) shows the effect of large value of τ in upper-bound model. The purpose of this figure is to show the effect of ideal thread execution



(a)



(b)

Figure 5.9: (a) Upper- and lower-bound surfaces of the composite prediction model associated with Eqs. 5.9 and 5.10 respectively for multi-core systems with $r = 4$ and $n = 16$. (b) Same surface diagram of the composite upper- and lower-bound model in an alternative perspective.

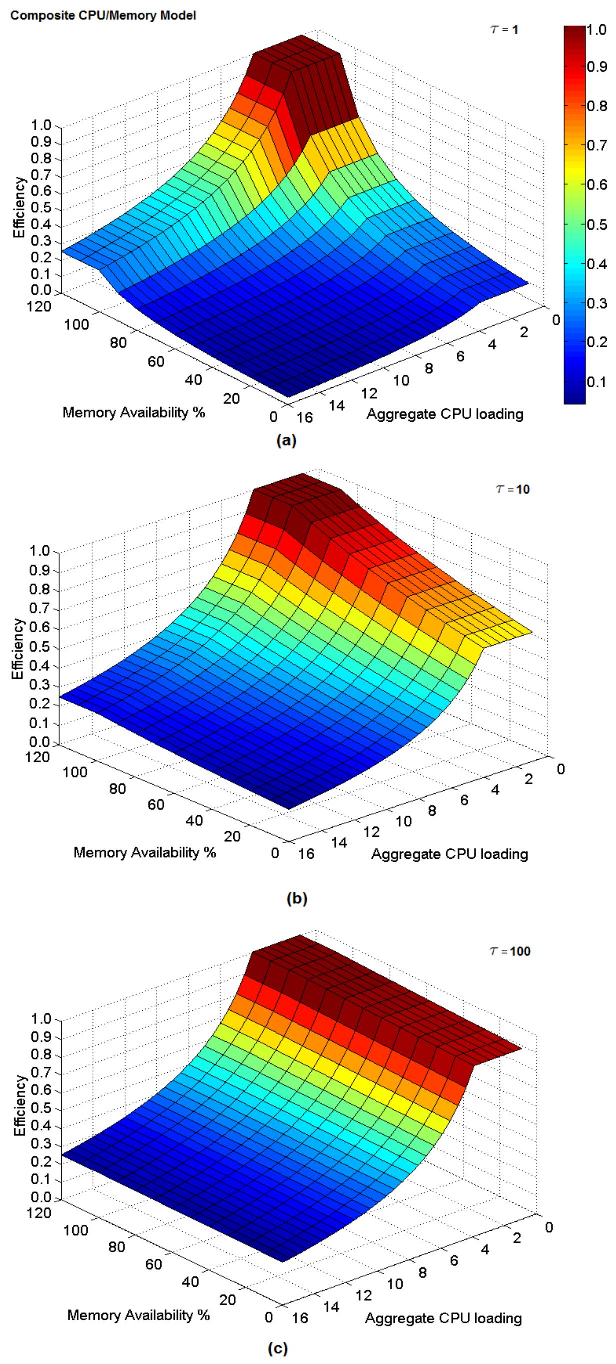


Figure 5.10: Upper-bound surface diagram for the composite prediction model associated with Eqs. 5.9 with $r = 4$ and $n = 16$ (a) when $\tau = 1$ sec (b) when $\tau = 10$ sec, and (c) when $\tau = 100$ sec.

time (τ) when the value changes from 1 to 100 sec. It can be observed among surfaces that when the value of τ increases from 1 to 100, the overall efficiency of the node increases considering the memory requirement is same in all three cases. The efficiency value increases because when the program runs for longer period of time, the page fault overhead (which is same for all the three cases) has less impact compared to a program which runs for a shorter period of time. A similar effect is illustrated in Figure 5.1 of Section 5.1.2.

In the following section, experimental studies are performed to measure processor efficiency while executing threads to verify that the introduced upper- and lower-bound composite models can estimate the actual empirical measured efficiency surface.

5.2.3 Experimental Studies

5.2.3.1 Overview

The benchmark programs used for measuring the overall efficiency of a machine are: *Bitonic Sorting Network*, *Sparse matrix vector multiplication*, and *Tridiagonal Solver (using Gaussian elimination).* These benchmark programs consists of expressions that are mix of CPU and memory related operations and which makes them ideal for the composite model empirical studies. Similar approaches are taken for modeling threads by a series of alternating work and sleep phases as described in Section 3.3. The purpose of the experimental study is to empirically measure the efficiency of the machine as a function of aggregate CPU loading and memory availability factors. Aggregate CPU loading and memory requirement values are selected randomly and distributed among threads by using Algorithm 1, 2, and 3, respectively. Uniform sampling of data across the values of possible aggregate CPU loading and memory availability has been carried-out. For implementing the benchmark programs and composite model framework programs, most of the modules of the CPU and memory availability programs are reused.

Algorithm 4 Estimating thread execution performance spawned in a multi-core system.

```
Input: Number of threads ($n$), and test run ($tr$)
for count ← 1 ... $tr$ do
  Select a random aggregate CPU load value between [$\epsilon$ ... $n$]
  while $i \leq (n-1)$ do
    $UR = Min\left(L - \sum_{j=1}^{i-1} T_j - (n-i) \times \epsilon), 1\right)$
    $LR = Max\left(L - \sum_{j=1}^{i-1} T_j - (n-i)), \epsilon\right)$
    $T_i \in [LR, UR]$
    $T_i \leftarrow (UR - LR) \times T_i + LR$
  end while
  $T_n \leftarrow L - \sum_{i=1}^{n-1} T_i$
  Compute aggregate work for each thread using Eq. 3.6
  Compute sleep phase length for each thread:
    $Sleep = \left(\frac{StageWorkTime \times (1 - CPUusage)}{CPUusage}\right)$
  Use fork() to generate a thread to do the followings:
    Select a random MAP ranging from 1...120%
    Compute $R$, $M_a$, and $cm$ for selected MAP
    Allocated and initialize memory based on MAP
    Occupy and hold memory until all child threads finishes assigned tasks
      and terminate
  Create a new thread to control next level of child threads
  for $i \leftarrow 1$ ... $n$ do
    Create thread $t_i$, and pipe link with its parent
    Generate a large 2D array of computed size, depending on memory requirements
    Allocate and Initialize memory (to avoid fake allocation) for the array
    Replace page and compute the overhead, $\varphi$, as:
      $\varphi = \left(\frac{\sum_{i=1}^{n} tm_i}{page\ size} \times \sigma\right) \times \omega$
    Release the memory (i.e., demalloc) occupied by the array
    Take the end time snapshot for the thread $t_i$
    Compute the backing store (secondary memory) overhead,$\eta$, as:
      $\eta = \left(B_l + \sum_{M_a+1}^{R} (B_s) + \kappa\right)$
    Compute efficiency, upper, and lower bound
    Transfer test case values to its parent via pipe
  end for
  Wait for all child threads signal to terminate
  Persist the measured efficiency data in respective CSV files
end for
Close all files and connections
Output: Thread execution data for statistical analysis
```

About $18,000$ test runs are conducted for three independent sets of test cases in which 8, 12, and 16 threads are spawned concurrently in a quad-core machine. This vast number of test runs are conducted to sufficiently cover the possible scenarios of thread executions. A similar experimental environment as described in Section 4.2.2 has also been deployed here for conducting the case studies.

5.2.3.2 Benchmarking

A similar but combined approach has been performed (described in Sections 3.6.3 and 5.1.5.3) for setting up the empirical framework programs of the composite model. For the memory portion, each thread was assigned an array size of 50MB and the data is generated randomly for initialize the array. For the CPU portion, sleep phase length and total work load values are calculated by utilizing Eqs. 3.5 and 3.6, respectively. The thread with the minimum CPU load was assigned 1.0×10^8 units of work load, which should be accomplished in 25 work-sleep phases. In the work component of each phase, threads accomplish 4.0×10^6 units of fixed computational work. Thus, when the memory requirements of threads is less than the available memory of the system, Eq. 5.11 can be used to determine the ideal thread execution time, denoted by ν. Input parameters for computing ideal execution time (ν) are work phase time in the quantum (Q_w), number of work phases to complete the total work (N_w), and CPU usage factor of the thread (L_i).

$$\nu = \left((Q_w \times N_w) + (N_w - 1.0) \times \left(\tfrac{Q_w \times (1.0 - L_i)}{L_i}\right)\right) \tag{5.11}$$

According to Eq. 5.11, the value of ν can be measured dynamically using the work phase length (measured by multiple test runs), number of work phases (total work/work accomplished in a single phase), and the CPU usage factor of the thread. All parameter values of Eq. 5.11 are available while the program is running. As the total amount of work depends on the CPU load of threads, the variable ideal

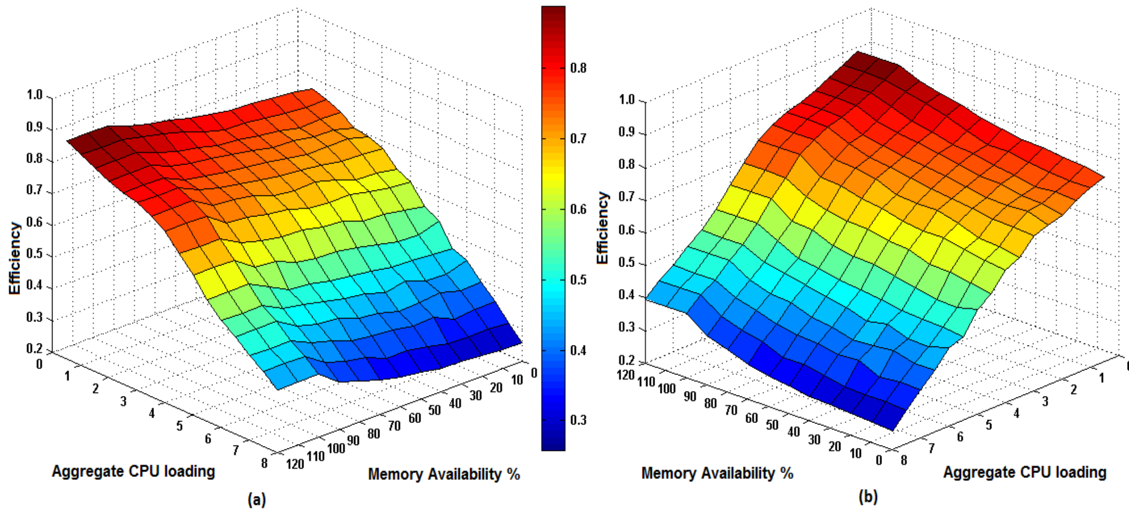


Figure 5.11: Measured efficiency surface of 8-threads in a quad core machine (a) when $\tau = 30$ and number of test runs are $4,000$. (b) The same surfaces in an alternative perspective for clear visualization of the effect of efficiency degradation due to memory availability.

thread execution time for threads (because of variable work load) can be measured accurately by utilizing the Eq. 5.11.

5.2.4 Case Study Results

For measuring the overall efficiency value of thread execution, three independent case studies were conducted on a quad-core machine for 8, 12, and 16 threads. Aggregate CPU load and memory availability values were selected randomly and distributed among the threads as described in Sections 3.6.1 and 5.1.5, respectively. As the focus of this empirical study is to measure the effect in overall efficiency of machines when both CPU and memory usage are varied, the number of threads deployed is always above the number of CPU cores of the machine and the memory availability is varied from 120% down to 0%.

Figure 5.12 (a) shows measured efficiency surface for the execution of 8 threads in a quad-core machine. About $4,000$ independent test cases are carried out to capture all possible execution efficiency scenarios due to CPU and memory availability variation. A moving average is taken from these test results with a sliding window size of 0.10 aggregate CPU loading and 0.5% of memory availability, and incremental value of 0.01 CPU loading and 0.5% memory availability. The data is then converted to a two dimensional matrix format for plotting a 3D surface diagram.

It can be observed from Figure 5.11 (a) is that the efficiency value decreases significantly when the aggregate CPU load reaches beyond the CPU cores (here, $r = 4$) because of the increased CPU contention among running threads. Figure 5.12 (b) illustrates the decrease of machine efficiency because of the decrease of memory availability. The efficiency decrease due to memory availability is moderate because the τ value of test runs are large (refer to Figure 5.1; here, $\tau = 30$ sec).

Figure 5.12 (a) shows the same surface diagram illustrated in Figure 5.11 (a). The same figure is converted in a relatively light color to differentiate from upper- and lower-bound surfaces to avoid any confusion while superimposing. In Figures 5.12 (b) and (c), the efficiency surface is superimposed with upper- and lower-bound surfaces associated with Eqs. 5.9 and 5.10, respectively. The purpose of Figures 5.12 (b) and (c) is to verify that the efficiency surface shown in Figures 5.12 (a) is bound using the composite upper- and lower-bound models. From the empirical results and measured efficiency surface plot in Figure 5.12 (b), it is apparent that theoretically derived upper- and lower-limits introduced in this section do bound the actual measured efficiency surface very well (Note: the efficiency surface diagram is given a different and light color so that any crossings can be detected immediately). It can also be seen from Figure 5.12 (b) that bounds are tighter when the aggregate CPU and memory loading is either relatively low or relatively high. Figure 5.12 (c) shows the same surface diagrams in a different perspective to illustrate that there are no

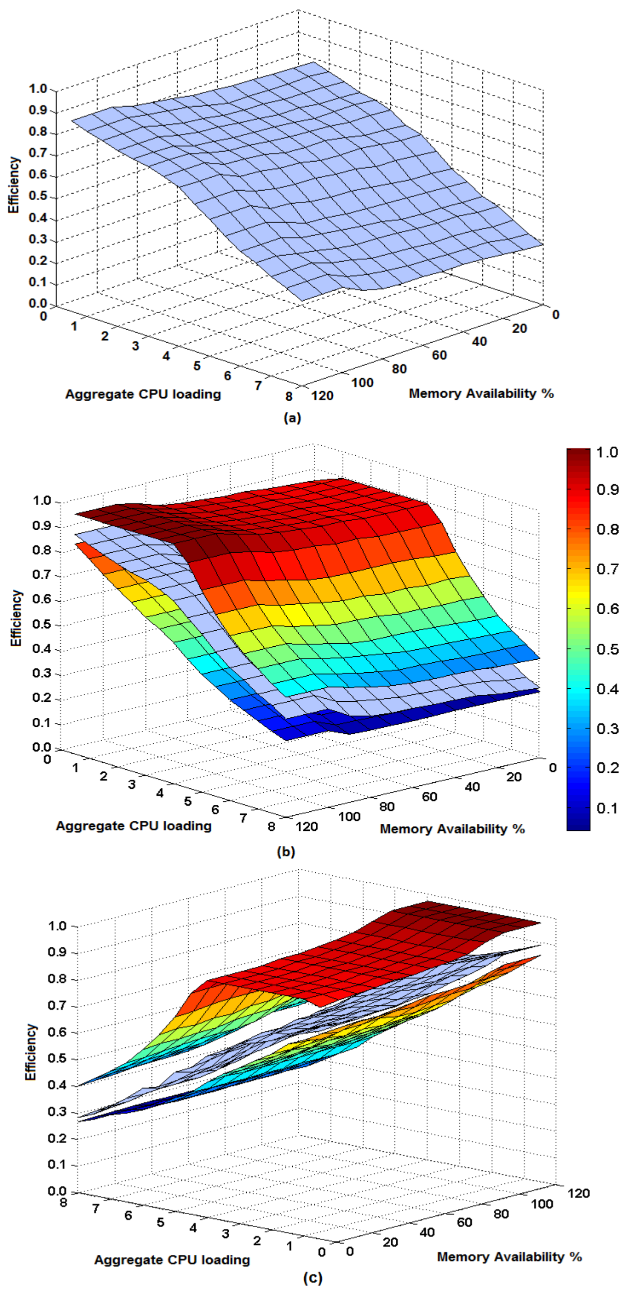


Figure 5.12: (a) Measured efficiency surface of 8-threads in a quad core machine for $\tau = 30$ and number of test runs are $4,000$. (b) The measured efficiency surface is superimposed with composite upper- and lower-bound surfaces. (c) Same surfaces in an alternative perspective for clear visualization of the binding of the efficiency surface by upper- and lower-model surfaces.

crossings among measured efficiency, upper-, and lower-bound surfaces. This figure also illustrates the degradation in efficiency when the memory availability decreases.

In the next set of test cases, 16-threads were spawned concurrently for measuring the overall machine efficiency of thread execution. A similar approach has been taken, like 8-threads, for conducting the empirical case studies. About 8,000 independent test cases were conducted in which CPU and memory availability were selected randomly and distributed among threads to cover all possible scenarios. Moving averages are taken from these empirical studies resulting data with a similar sliding window approach for generating efficiency surface data. Upper- and lower-bound surface data are generated from the same model Eqs. 5.9 and 5.10 respectively.

Figure 5.13 (a) shows the average measured efficiency surface for 16-threads in a batch. A similar decrease in efficiency has also been observed when CPU loading increases and memory availability decreases. Figures 5.13 (b) and (c) illustrates the efficiency surface superimposed with theoretically derived upper- and lower-bound surfaces. From Figures 5.13 (b) and (c), it is apparent that the upper- and lower-bound prediction models introduced here can also bound the actual measured efficiency surface very well.

5.3 Summary

Analytical models (and empirical studies) were developed for predicting (and measuring) execution efficiency of concurrent threads in Linux environment as a function of memory availability. As would be expected, degradation in thread execution efficiency occurs when primary memory availability is less than the total required memory by threads. In addition to memory availability, the total amount of memory required by concurrent threads is a factor in predicting the thread execution efficiency. When the total memory requirement is higher than the total available memory, the relative performance can have a significant impact on execution time.

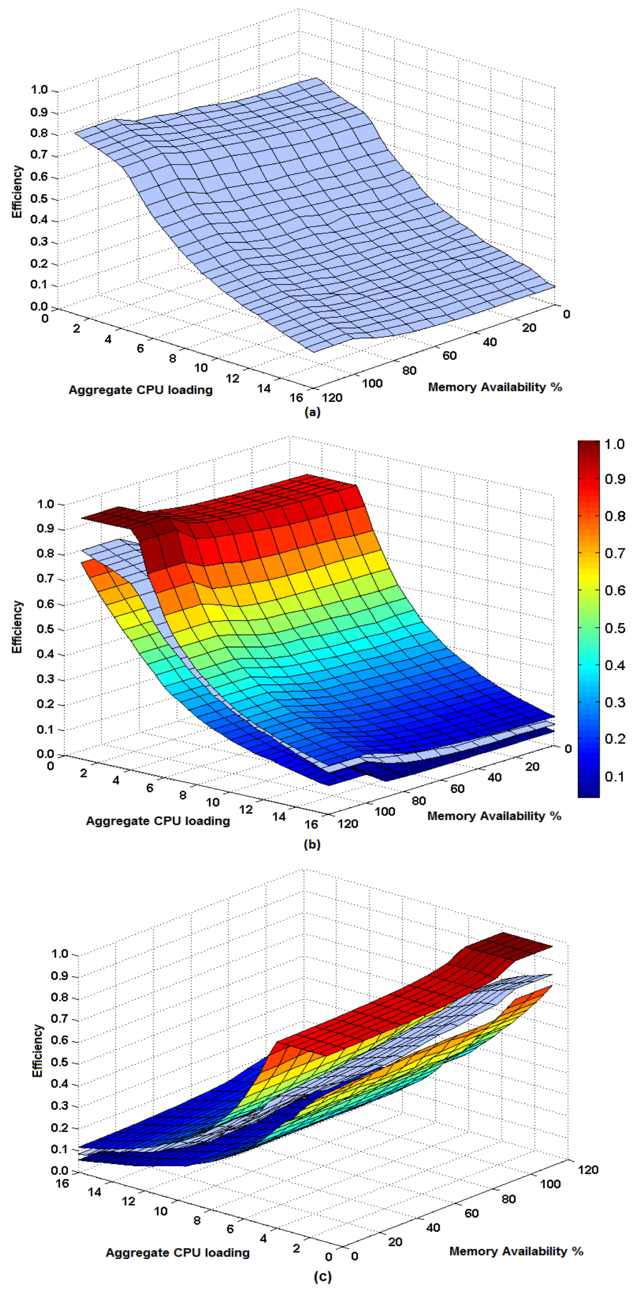


Figure 5.13: (a) Measured efficiency surface of 16-threads in a quad-core machine for $\tau = 30$ and number of test runs are $8,000$. (b) The measured efficiency surface is superimposed with composite upper- and lower-bound surfaces. (c) Same surfaces in an alternative perspective for clear visualization of upper- and lower-limit binding of the efficiency surface.

Specifically, increased page swap out by concurrent threads and long queue delay due to serial I/O degrades thread execution efficiency. A memory availability scatter plot provides a clear visualization of measured efficiency based on the density of the dots in the plot. These empirically measured availability values are showed to generally fall within theoretically derived upper- and lower-bound models.

The composite prediction model proposed in this chapter is derived from the CPU availability and memory models. It is necessary to have accurate model for estimating the overall machines' efficiency for concurrent thread execution on a time-shared system. The prediction models were validated empirically by an extensive set of case studies involving real-world benchmark programs having dynamic behaviors. The results of empirical studies are presented in this chapter in the form of scatter plots, surfaces, and tables. For the composite prediction model, surface plots for measured thread execution efficiency along with upper- and lower-bound surfaces provides a clear visualization of all possible cases of combined CPU and memory availability variation in thread execution. These empirically measured availability values are shown to generally fall within theoretically derived upper- and lower-bound models.

The proposed new prediction models can be highly useful for determining the order in which tasks should be assigned to the system so that the completion time of all the tasks is minimized. The composite prediction model can be used as a building block for distributed task scheduler (or for allocating real-time resources in cloud environment) to determine the order (or find sub sets) in which tasks should be assigned to compute nodes for minimizing the total execution time prior to the placement of tasks into the run queue.

Chapter 6

Task Assignment Policies

In this chapter, *two new task assignment policies* are introduced by utilizing the composite prediction model presented in Chapter 5. The goal of this chapter is to find an assignment of tasks to compute nodes such that the total time taken to execute all the tasks is minimized. The composite prediction model provides required intelligence along with a new CPU availability prediction model to make proper task assignment decisions. The new CPU usage prediction model is capable of estimating available CPU resources for a new task before placing it into the run-queue. This new model provides task scheduler the benefit to look-ahead and find the best arrangement for a set of tasks to minimize total execution time. The introduced task assignment policies are then compared empirically with the popular Terascale Open-source Resource and QUEue Manager (TORQUE) for the purpose of measuring the task assignment performance difference.

6.1 Introduction

System performance prediction involves estimating the system's behavior for the set of tasks to be executed on it. The prediction of resource availability in a system is important in the context of making task assignment, load balancing, and scheduling decisions in distributed systems. Making such predictions is complicated by the dynamic nature of the system and its workload, which can vary drastically in a short span of time [1]. For any prediction approach, it can be useful to know *a priori* certain characteristics of tasks that are planned to be assigned to the system. As an example, it is useful to know the maximum amount of main memory a task will consume during its execution (*memory requirement*). This information is useful to forecast the amount of time that may be consumed for memory paging activities. It is also useful to know the *CPU requirement* of the task, which is the fraction of time a task requires the CPU.

For tasks such as generating prime numbers, computing Fast Fourier Transforms, and related others require significant use of the CPU. One can also know the CPU and memory requirements of a task based on the information gathered from its earlier executions. The execution of many scientific models (economic, meteorological, and others) fall into this category, where the program remains the same and the data on which it operates changes over time.

Given the CPU requirements of tasks in a run queue, Beltrań et al. [1] provide an analytical model to estimate the *CPU availability* (which is the percentage of CPU time that will be allocated) for the new task prior to its placement in the run queue. It is shown that in certain cases this information can be used to schedule the execution of tasks in such a way that the completion time of all the tasks is minimized [1]. Khondker et al. [7] extended the work of Beltrań et al. [1] as follows. Beltrań considered a batch of tasks (each with its own CPU requirements) and their analytical model determined the CPU availability using the *sum total* of the CPU requirement of each of the tasks in the batch. Using this sum total posed a challenge in that the CPU availability prediction is precise only when the order of task execution is known *a priori.* To address this challenge, the analytical model in Khondker provides tight upper- and lower-bounds on CPU availability. The bounds are necessary because the actual CPU availability depends on the order of execution of tasks in the batch. Thus the analytical model in Chapter 3 is oblivious of the CPU scheduler.

In this chapter, we have further improved the analytical model in [7] in several ways. First, we utilize the *memory model* that determines the *execution efficiency*, which is the amount of CPU time that a task requires to complete its execution when adequate free memory frames are available in main memory divided by the total time it takes when only part of the necessary memory frames are available in the main memory. Second, we have utilized the *composite model* which is derived from the CPU availability model in Section 3 and the memory model developed in Chapter 5 to derive a *composite model.* This prediction model consists of analytically derived upper and lower efficiency bounds helps to determine the best efficient node in the system. Third, we have introduced a *new CPU usage* prediction model for measuring CPU usage before assigning a task in to the machine. In this empirical model, we calculated the new CPU usage of the machine based on the CPU availability of each core and the CPU requirements of the task. Moreover, the proposed model considers multi-core CPUs and hyper-threading for accurate prediction. Additionally, this new model is able to predict the *CPU availability for a new task* without explicit knowledge of the mapping between available cores and tasks. Fourth, two new *task assignment policies* are proposed by utilizing the composite and new CPU usage models for dispatching tasks intelligently in distributed multi-core compute nodes. Finally, a *task look-ahead* approach is proposed for a set of incoming tasks from the run-queue by utilizing the new CPU usage model.

Using the availability models developed in Chapter 3, 4, and Chapter 5, we have designed algorithms for task assignment in distributed multi-core compute nodes. Given a set of tasks (with known CPU and memory requirements) and set of compute nodes, we use the composite model to determine the best node (in terms of thread execution efficiency) and assign the task to that node. We have provided two approaches for task assignment to a compute node using the composite prediction model. In the first approach named Task Assignment using Status Measurement (TASM), we query all of the nodes each time a task is to be assigned to gather the nodes' status information and use the composite model for decision making. While such current information give us precise input to the models, it can incur high overhead when we consider a large number of compute nodes. To overcome this challenge, we have implemented a second approach, wherein we only know the initial state of the compute nodes and the status of the compute nodes is dynamically updated as we assign tasks to them using the prediction models developed in this paper. We call this second approach Task Assignment using Status Estimates (TASE). Our extensive empirical evaluations have shown that TASE performs equally well (in terms of total completion time) in comparison with TASM model, thereby avoiding the overhead of querying the nodes for their status each time a task is to be assigned.

A number of scenarios using real benchmark test loads with dynamic behavior are carried out to measure the accuracy of task scheduling models. We have compared our introduced task models with the widely used Terascale Open-source Resource and QUEue Manager (TORQUE) for performance comparison. In addition, extensive empirical work is conducted to measure the accuracy of the proposed composite prediction model, and the new CPU usage prediction model as well which are the building blocks of the task models.

The rest of the chapter is organized in the following manner. Section 6.2 presents the Terascale Open-source Resource and QUEue Manager (TORQUE) and it's job scheduling policies. Two task assignment models along with the task dispatch algorithm are presented in Section 6.3. Section 6.4 provides information of the empirical environment, benchmark programs along with case study results for both task assignment models and compares the achieved performance with the popular TORQUE system. Section 6.5 illustrates task look-ahead strategy for the TASE model along

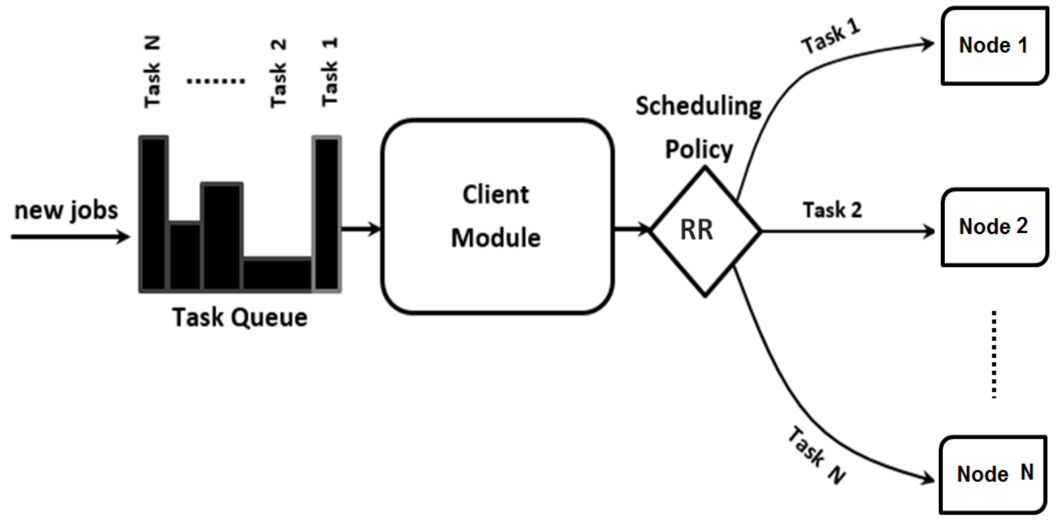


Figure 6.1: A high-level task scheduling diagram for the TORQUE system which uses Round Robin policy to assign batch job for distributed compute nodes.

with empirical results to demonstrate the importance of task execution order and accuracy of the model. Finally, Section 6.6 summarized contributions and application areas for the introduced models.

6.2 TORQUE Distributed Resource Manager

The Terascale Open-source Resource and QUEue Manager (TORQUE) is a *distributed resource manager* that provides control over batch jobs and distributed compute nodes. TORQUE is an open-source distributed resource manager software and freely used, modified, and distributed [55]. It incorporates significant advances in the areas of scalability, reliability, and functionality and is currently in use at many leading government, academic, and commercial sites throughout the world [56].

Because of the availability of TORQUE scheduler protocol and wide spread use of the product, TORQUE scheduler is considered as a benchmark to evaluate new task assignment models. In the TORQUE task assignment model, all incoming tasks are placed in a task queue. The task queue operates in FCFS order. Figure 6.1 shows the basic high-level diagram of the TORQUE system. Incoming jobs are placed at the end of the task queue and removed from the head. That is, it uses First Come First Serve (FCFS) queuing approach when scheduling jobs on the distributed computing resources. Jobs are then dispatched to distributed compute node using Round Robin (RR) scheduling technique [21].

6.3 Task Assignment Models

In this section, two new task assignment models in distributed environment have been introduced by utilizing the composite prediction model. The goal is to find an assignment of tasks to compute nodes such that the total time taken to execute a batch of tasks is minimized. For updating the CPU resource availability in real-time while tasks are assigned to distributed compute nodes, a new CPU availability prediction model is introduced which can predict the resource after each task dispatch. Note, the CPU availability model introduced in Chapter 3 is different form the model introduced in this chapter which provides an estimated CPU availability before each task assignment instead of an upper- and lower-bound CPU availability estimate for a batch of tasks.

6.3.1 Task Assignment using Status Measurement (TASM) Model

In this task assignment model, all incoming tasks are placed into a task queue. The task queue operates in FCFS order like TORQUE. Figure 6.2 shows the task distribution system for the TASM model. A client-server methodology has been adapted to design and implement the analytical framework for the task assignment model. In this distributed model, a client program initially retrieves IP addresses and port numbers of all servers and sends resource status request. For assigning tasks

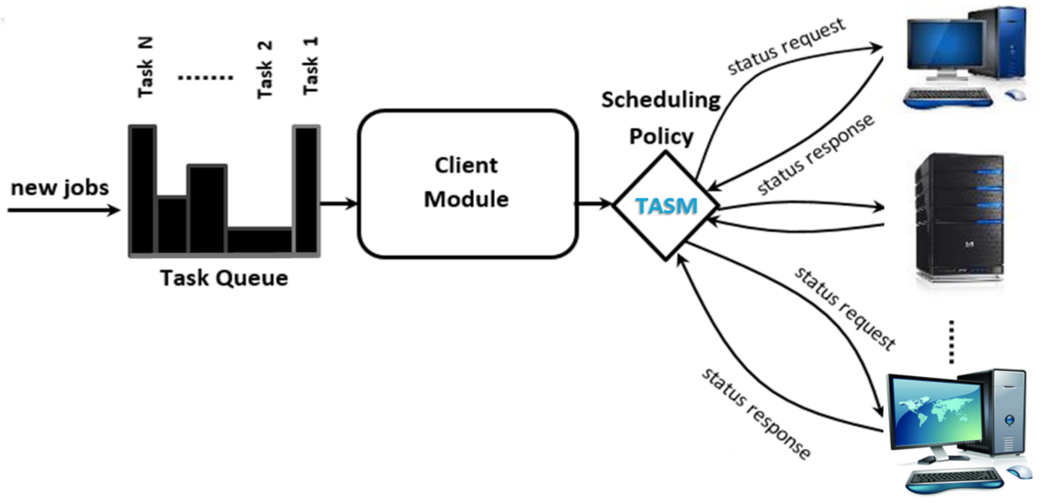
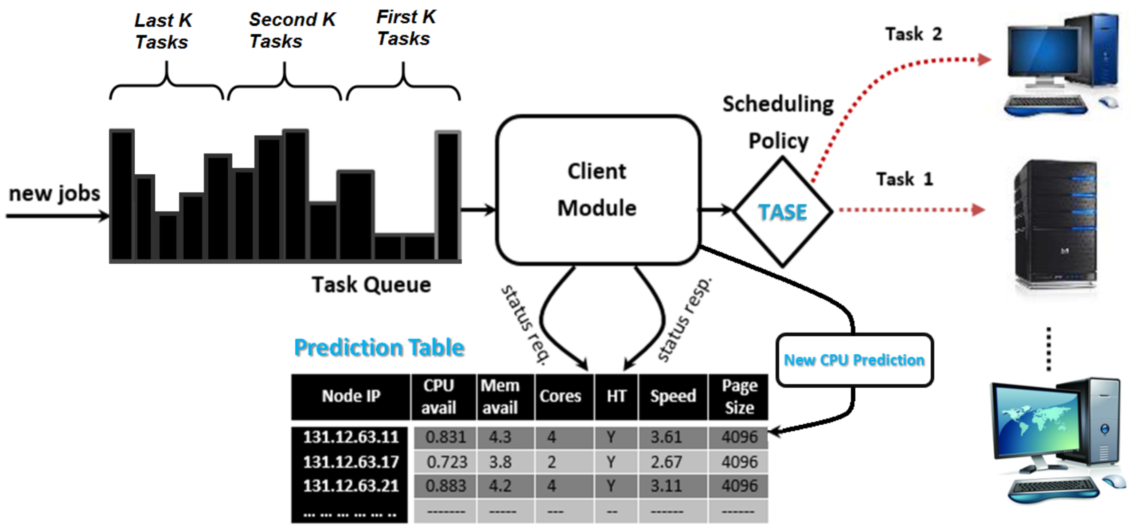


Figure 6.2: Task assignment to distributed compute nodes using the status data of compute nodes.

to compute nodes, a task is popped out from the head of the queue. Properties of the task and state information of each compute node are collected. Properties of a task includes CPU and memory requirements of the task. State information of a node includes resource information including current CPU usage/load, available free main memory, page size, number of cores, and maximum speed of cores. Then the composite prediction model is deployed to measure the execution efficiency of the task for all compute nodes. The task is then assigned to the most efficient node. State information of all nodes in the grid are collected each time a task is to be assigned. This method provides most current and precise state information to the models but expensive, in terms of overhead, when we consider a large number of compute nodes.

6.3.2 Task Assignment using Status Estimates (TASE) Model

In this model, a dynamic prediction table is maintained for holding the predicted CPU and memory availability data of each node of the distributed system. The



Node IP	CPU avail	Mem avail	Cores	HT	Speed	Page Size
131.12.63.11	0.831	4.3	4	Y	3.61	4096
131.12.63.17	0.723	3.8	2	Y	2.67	4096
131.12.63.21	0.883	4.2	4	Y	3.11	4096
... ...	-------	-----	---	--	------	------

Figure 6.3: Task assignment to distributed compute nodes using a *dynamic prediction table* consists of real-time prediction of status data of compute nodes.

same composite prediction model of TASM is used but the resource state parameter values are taken from the prediction table instead of fetching it from nodes each time before task assignment. The prediction table of this model is initially populated by queuing resource status information from each computing nodes. After that, the resource status in the table is updated dynamically using CPU usage prediction model. The CPU usage model updates the table each time a new task is assigned to a node or an old task terminates and release resources.

Figure 6.3 shows the task distribution system for the TASE model. A client-server methodology has been adapted to design and implement the analytical framework for both task scheduling models. In this distributed model, a client program initially retrieves IP addresses and port numbers of all servers and sends resource status request. Upon arrival of status, it's then used to populate the prediction table in TASE model. The client program waits for service request like *show task queue*, *show grid state*, *dispatch tasks to nodes*, *show task processing status*, *show task execution statistics*, etc. When a task dispatch request is received, the client program generates a thread using the function $fork()$ to handle the task dispatch service based on introduced task assignment models. After assigning the task dispatch job to the newly created thread, the client program goes back and waits for another service request.

6.3.2.1 CPU Usage Prediction Model for a New Task

In this section, two prediction models are introduced to measure the *CPU usage of a compute node* and *CPU availability for a new task* prior to the placement of the task into the run queue. Parameters to these models are the CPU requirements of a new task, number of cores, current CPU usage of cores, degree of hyper-threading, and others. By utilizing these parameters, the CPU usage prediction model facilitates users to measure new CPU usage of a compute node before assigning a task into it. From the predicted new CPU usage and current CPU usage, a new model is derived to estimate CPU availability for a new task. Additionally, this new model is able to predict CPU availability without explicit knowledge of the mapping between available cores and tasks.

Existence of the new CPU usage prediction model is important because there exists many cases in which an average CPU load value (can be extracted from the kernel) will not be able to reflect the actual CPU availability for a new incoming task. For example, consider two dual-core machines M_1 and M_2 with same configuration. Further consider, the M_1 is running 2 I/O bound-tasks containing CPU requirement of 0.4 and 0.5 respectively. The M_2 is running only 1 CPU bound task with 0.9 CPU requirement. In both cases, the average CPU load will be 0.45 (equally loaded). Now, for a new incoming task with CPU requirement of 0.8, the M_2 has better capability of providing faster run-time compared with the M_1 because one core of M_2 is unoccupied. The new task can get full attention of a core compared with the M_1 in which each core is preoccupied with a task.

As illustrated in previous subsection (and in [7]), exact values of CPU availability are difficult to predict because of dependencies on many factors, including context switching overhead, memory speed, CPU usage requirements of the threads, core hyper-threading, the degree of interleaving of the timing of the CPU requirements of the threads, and the characteristics of the thread scheduler of the underlying operating system. Due to the complex nature of the execution environment, an approach is employed here to estimate expected CPU availability.

The proposed new CPU usage prediction model considers the CPU load of each core of a machine instead of one CPU load value (aggregate) for the whole system to accurately predict the new CPU usage for an incoming task. From several empirical studies, it has been observed that a new task is placed in the core which has the minimum CPU usage (i.e., maximum CPU availability). Here, r is the number of physical cores, ξ represents the number of hyper threads in a core, and ϑ represents the CPU requirement of the thread. Additionally, C_l represents the logical core which has the minimum CPU load where $l \in \{0, 1, .., (r \times \xi) - 1\}$ and and $C_l = min(C_1, C_2, ..., C_{((r\times\xi)-1)})$. The new measured CPU load of the machine, denoted by α_p, can be represented as follows:

$$\alpha_p = \begin{cases} \frac{(1.0-C_l)\times\vartheta}{r\times(\xi-\vartheta)}, & \text{if } C_l = 0 \\ \frac{C_l}{(r\times\xi)} + \frac{(1.0-C_l)\times\vartheta}{r\times(\xi-\vartheta)}, & \text{else if } 0.0 < C_l < 1.0 \\ \frac{C_l}{(r\times\xi)}, & \text{else if } C_l = 1.0. \end{cases} \tag{6.1}$$

Eq. 6.1 is based on the concept of utilizing the benefits of hyper-threading. For pure CPU bound tasks, hyper-threading technique brings slim benefit compared to I/O-bound tasks. Hyper-threading can facilitate the overlap of I/O and computation. If multiple processors are available, I/O-bound threaded applications may see substantial speedup. Therefore, the model provides more available CPU to I/O-bound tasks by dividing the predicted core load using $r \times (\xi - \vartheta)$ and CPU-bound thread using $(r \times \xi)$.

The CPU usage of the compute node, denoted by P_c, represents the new measured CPU load of the machine and expressed as:

$$P_c = \frac{\sum_{i=0, i \neq l}^{n-1} C_i + \alpha_p}{(r \times \xi)}. \tag{6.2}$$

In this empirical model, we calculate the new CPU usage of the machine based on the CPU availability of each core and the CPU requirements of the task. Moreover, the proposed model considers multi-core CPUs and hyper-threading for accurate prediction. The CPU availability prediction for a new task, denoted by P_t, can be measures by the following expression.

$$P_t = \left\{ \left(\frac{\sum_{i=0}^{n-1} C_i}{(r \times \xi)} \right) - P_c \right\} \times (\xi - \vartheta). \tag{6.3}$$

Eq. 6.3 provides a model/explanation of what has been observed for thread assignment in processor in [7]. Empirical studies for verifying the accuracy of prediction models presented in this section are presented in Section 6.4.3. The new CPU usage prediction model, expressed in Eq. 6.2, is implemented and used in TASE model to re-calculate the CPU usage in prediction table for the most efficient node in which the task was dispatched.

6.4 Empirical Studies

Two sets of empirical studies are carried out for validating the CPU usage prediction model, and task scheduling model. As both composite prediction and CPU usage models are the building block of the two task scheduling models, we have conducted an extensive set of empirical studies to validate the accuracy of proposed prediction models. Thereafter, we have conduct a second set of empirical studies to depict the effectiveness of both TASM and TASE task scheduling models by comparing task completion times with standard Round Robin (RR) scheduling using FCFS queue (used by TORQUE system [21]).

6.4.1 Overview

The execution environment we have utilized is a computer grid in which clients can accept/generate and request tasks for execution. The environment consists of a set of servers $\{S_1, S_2, .., S_n\}$ where each server controls a local execution resource that allows execution of one or more tasks. The server controls execution resources with different number of cores. The servers are connected by a grid network such that each server has a connection to a set of neighboring servers. Task requests (generated by clients) are placed in a FCFS queue for dispatch. Introduced prediction based task scheduling models are deployed to select the best server node for tasks in the queue. A server receiving task execution request is responsible for the execution. After completion of execution, task run statistics along with a feedback message is sent back to the client.

Algorithm 5 shows the high-level pseudo code of the client model which includes three studied task assignment models. The composite prediction model takes task and node properties as input and provides the IP address of the most efficient node for the task. After task completion, the server side program send the task execution statistics back to client program. The client program stores the results in various files and updates the task processing status. The next task is chosen from the head of the task queue and the above method is again followed.

The server side program is responsible for sending the resource status and task execution statistical information. As soon as the server program receives a message, it checks the type of the message. There are two major types of messages, namely *resource status* and *task execution*. If the message is a resource status request then Unix script files and system functions like */sys/devices/*, */proc/stat/*, and *sysinfo()* are utilized to retrieve *utime*, *stime*, *nicetime*, *idletime*, *pagesize*, *free memory*, and *number of running processes* information. This information is then converted to a usable format and sent back to the client program.

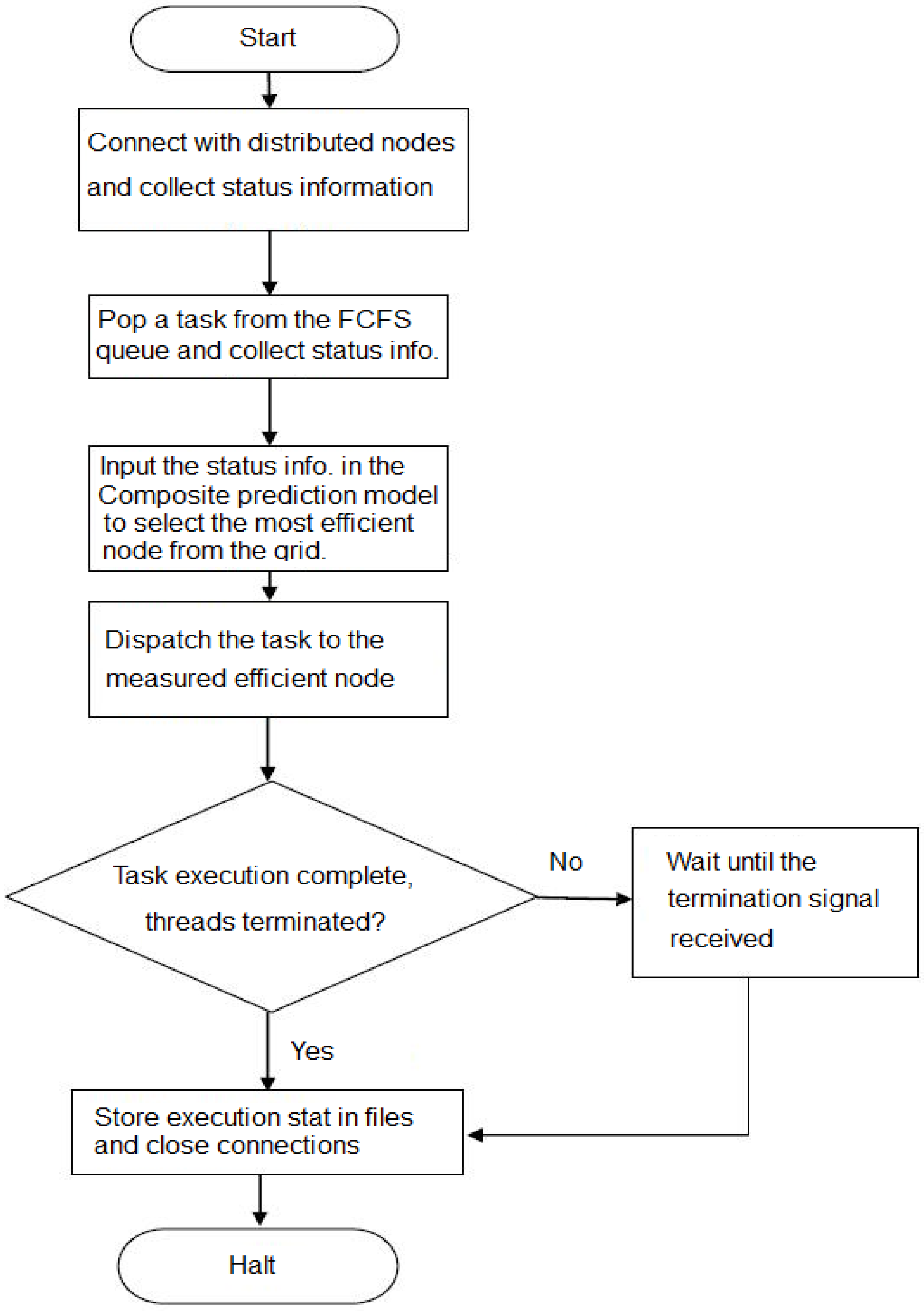

Start
Connect with distributed nodes and collect status information
Pop a task from the FCFS queue and collect status info.
Input the status info. in the Composite prediction model to select the most efficient node from the grid.
Dispatch the task to the measured efficient node
Task execution complete, threads terminated?
No
Wait until the termination signal received
Yes
Store execution stat in files and close connections
Halt

Figure 6.4: A high-level flow diagram showing the task assignment steps for the TASM model.

If a server receives a task execution request, in both models, a child thread is created to carry out the operation so that the server can handle resource status and multiple task execution request in *parallel*. A task can be either synthetic or real benchmark programs. If the child thread received a synthetic task request, then a thread is created using $fork()$ function and task properties are passed into it. To validate the the CPU availability prediction and memory model, synthetic threads are used because they provide precise and parameterized control over test runs. If the child thread receives real benchmark task request, then $System()$ function is used to execute the task. Task execution statistics are then transfered to the server program via pipe communication. The server program sends back the execution statistics to client program via network socket.

6.4.2 Empirical Environment

The systems used for evaluating the task assignment models are three Intel(R) Xeon(R) Quad-core CPU W3520 @ 2.67GHz clock speed, $1,333$ MHz bus speed and 6.0 GB of RAM and one Intel dual core @ 3.06GHz clock speed, $1,333$ MHZ bus speed and 4.0 GB of RAM. These nodes are equipped with Linux kernel version $2.6.32-22$. The *average CPU load* (represents the average system load over a period of time) was 0.018136 per core in a scale of 1.0 (in a fifteen minute period) before running test cases which indicates that the nodes were lightly loaded (essentially unloaded). The *C programming* language was used to implement the prediction based resource management framework (analytical prediction and task assignment models) with the *gcc compiler* version 4.6.3.

Threads deployed for composite prediction model validation are independent tasks, meaning there are no interdependencies among threads such as message passing. Threads deployed for TASM and TASE models are real-world benchmark programs like *prime number generation, polynomial modular arithmetic, merge sort, image rendering* etc. Descriptions of benchmark tasks are provided in Table 6.1.

Algorithm 5 Task assignment using composite prediction model with FCFS queue.

```
Input: FCFS task queue ($\Phi$)
for i $\leftarrow$ 1 ... $nodeCount$ do
  Obtain the IP and Port number of Node $i$.
  Send resource status req. to Node $i$.
  Collect resource status and place into the prediction table.
end for
while true do
   Wait for service $request$ input.
   if ($request == taskDispatch$) then
      Create a thread using $fork()$ to handle task dispatch.
      if ($dispatchmethod == TASM$) then
      for t $\leftarrow$ 1 ... $taskCount$ do
         Pop a task from the head of the FCFS queue.
         Collect resource status for all nodes of the Grid.
         Evaluate the Composite prediction model Eqs. 3.1, 3.5.
         Select the node with maximum efficiency for the task.
      end for
      if ($dispatchmethod == TASE$) then
      for t $\leftarrow$ 1 ... $taskCount$ do
         Pop a task from the head of the FCFS queue.
         Collect resource status info. from the Prediction table.
         Evaluate the Composite prediction model Eqs. 5.9, 5.10.
         Select the node with maximum efficiency for the task.
         Re-calculate resource state of the node using Eq. 6.3.
      end for
      if ($dispatchmethod == RR$) then
      for t $\leftarrow$ 1 ... $taskCount$ do
         Pop a task from the head of the FCFS queue.
         Assign task using Round Robin scheduling.
      end for
   Spawn tasks concurrently to selected nodes using models.
   Update the task status to Processing.
   end if
   for t $\leftarrow$ 1 ... $taskCount$ do
      Read $task\ completion\ time$, $utime$, $CPU$, $and$
        $memory\ usage$.
      Update the prediction table after task completion.
      Update the task status to Complete.
end while
Output: $utime$, $total\ time$, and $CPU\ availability$
```

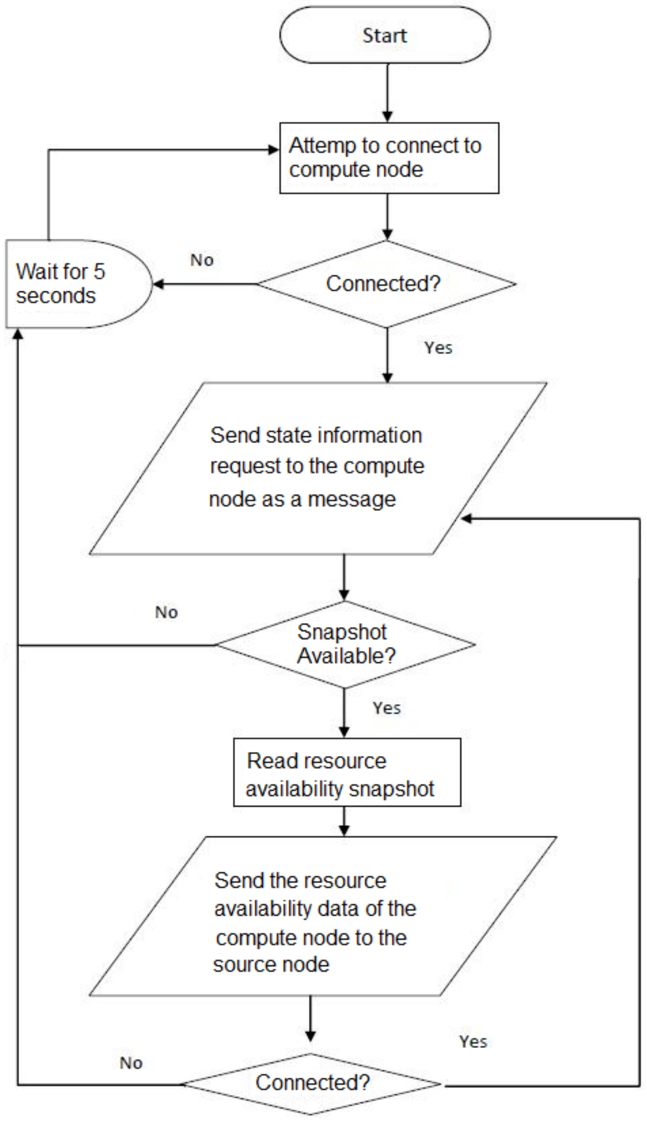

Start
Attemp to connect to compute node
Connected?
No
Yes
Wait for 5 seconds
Send state information request to the compute node as a message
Snapshot Available?
No
Yes
Read resource availability snapshot
Send the resource availability data of the compute node to the source node
Connected?
No
Yes

Figure 6.5: A flow chart showing the connection between source and compute node for task assignment models.

Table 6.1: Benchmark test programs along with associated loads for the Task scheduling model.

Name	**Description**	**Load Type**	**CPU Util.**	**Memory**
primeGen	Prime Number generator.	CPU-bound	1.0	Very Low
polyMod	Polynomial Modular Arithmetic.	Near CPU-bound	0.89	Low
mSort	Merge Sort Ordering Program.	IO-bound	0.63	Very High
linSer	Linear Search Program.	IO-bound	0.42	High
iRender	Image Rendering of Large BMP files using Image Smoothing Algorithm.	IO-bound	0.26	Moderate
swk	Synthetic work load.	IO-bound	0.13	Moderate

Threads are spawned concurrently in selected nodes based on the model's efficiency prediction. Additionally, CPU usage prediction model is required in TASE model for maintaining the prediction table dynamically after each task dispatch. When a task finishes its work, a task execution report is produced, which contains *task start time, task execution CPU time, idle time, end time, CPU availability for the task*, and *others.*

6.4.3 New CPU Usage Prediction Model Case Studies

The prime objective of this section is to measure the accuracy of the new CPU usage prediction model for a compute node. A CPU usage analytical framework has been developed to for estimating (and measuring) the new CPU usage of a multi-core machine before and after assigning new tasks. Different real world benchmark test programs shown in Table 6.1 with various CPU and memory requirements are selected as a group and used for the empirical study. These test loads are representative CPU-bound and I/O-bound tasks and their features are summarized in Table 6.1.

Algorithm 6 provides a high-level pseudo code for the CPU usage empirical framework. A batch set of tasks are placed into the run-queue with known CPU usage of each task. Model parameters are then initialized by the CPU availability of each core fetched from UNIX */proc/stat*. A child process is created to run each task and the task properties are send via *pipe* communication. Main program iterates back and creates a new child thread to handle the second task. Before running the second task, current CPU usage and model prediction (using Eqs. 6.1, and 6.2) are measured and sent back to parent thread. Parent thread then spawns second thread, and so on. Parent thread calculates the prediction error for all cases and stores the results in a CSV file for post-mortem analysis.

Algorithm 6 Prediction of CPU usage for a batch of tasks in the queue.

```
Input: Set of tasks ($N$) in the queue.
Measure the current CPU usage of each cores using CPU
  data from $/proc/stat$.
Initialize the model parameters ($e.g., numberofcores,$
  $coreload, totalload, andothers$)
for $i \leftarrow 1$ ... $N$ do
  Create a child thread using $fork()$ for running a task.
  if($childThread == true$) then
    Send task parameters via pipe
    Spawn the task $i$ into the machine
  end if
  Measure the CPU usage using model Eqs. 6.1 and 6.2.
  Measure the actual CPU usage.
end for
Collect the model and actual CPU usage using pipe.
Wait for all child threads to terminate.
Calculate the prediction error $|Actual - P_c| \times 100$.
Store measured data to respective files.
Output: Actual CPU Usage, Model Prediction Usage, and Prediction Error.
```

Figures 6.6, 6.7, and 6.8 correspond to experiments for a set of 12, 16, and 20 tasks in a run-queue for a quad-core machine. In the figures, the horizontal axis represents the number of tasks (N), and the vertical axis represents the CPU usage

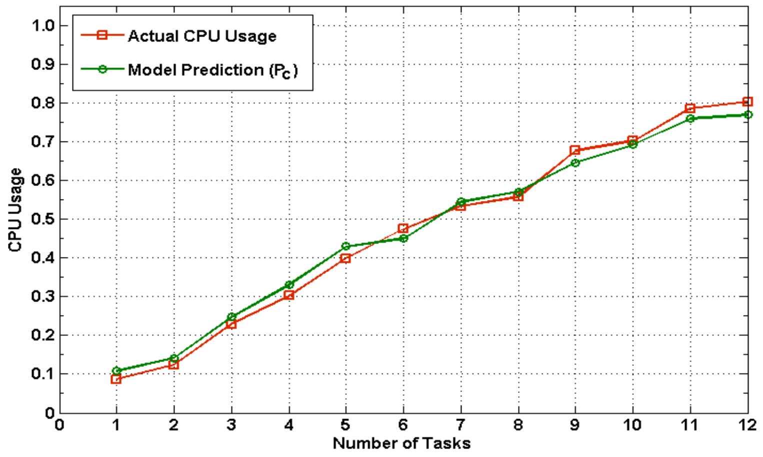


Figure 6.6: Comparison between actual measured CPU usage and the prediction model after dispatching 12 tasks concurrently.

of the machine. The actual CPU usage line is an average of multiple run. It can be observed from Figure 6.6 that the CPU usage prediction model line and the actual CPU usage line follows same basic shape and pattern. The variation between actual and predicted lines are very low. Figure 6.7, and 6.8 also depicts similar shape and patters between actual and predicted measures with low variation. The CPU usage prediction model lines are smooth compared to the actual CPU usage lines.

Table 6.2 shows results of spawning 20 tasks from task queue to a quad-core machine. For each of the cases, task name (Scenario), measure of actual CPU usage (U), measure of CPU usage prediction model (P_c), and the prediction error ($\delta_{P_c} = |P_c - U| \times 100$) are measured and provided in Table 6.2. The real-world benchmark programs listed in Table 6.1 are placed into the task queue and spawned concurrently. After dispatching each task, real CPU usage and predicted usage are measured. These two magnitudes are then used to compute prediction error percentage. it can be observed from the table that our CPU usage prediction model achieves very low prediction error. The minimum prediction error is 0.3% for the cases in which 1 and 15 tasks spawnned in the system. The maximum prediction error is 5.4% in which 13 tasks were dispatched. Regardless of CPU- or I/O-bound

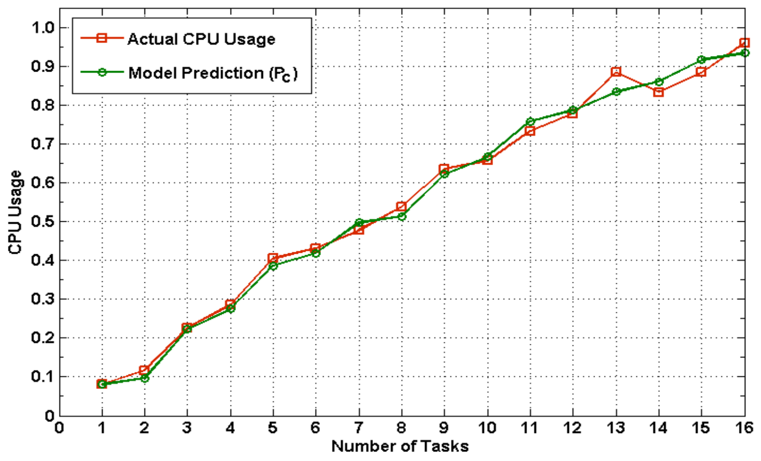


Figure 6.7: Comparison between actual measured CPU usage and the prediction model after dispatching 16 tasks concurrently.

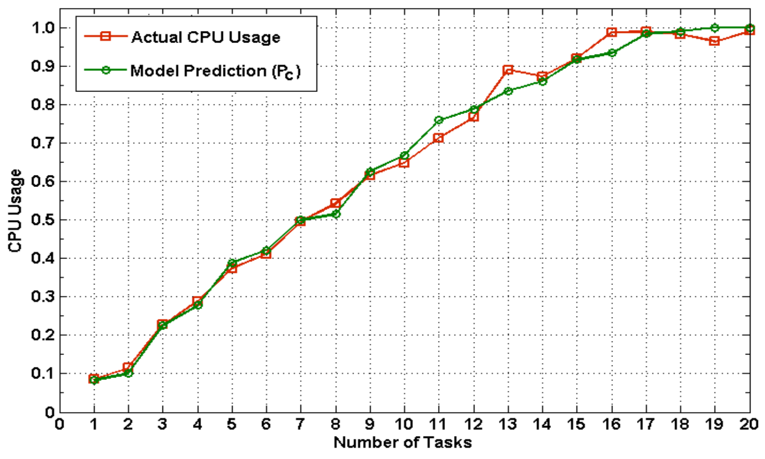


Figure 6.8: Comparison between actual measured CPU usage and the prediction model after dispatching 20 tasks concurrently.

Table 6.2: Comparison between actual CPU usage and the Prediction model.

N	**Scenario**	**Actual**	P_c	δ_{P_c}
1	mSort	0.086	0.083	0.30
2	mSort + iRender	0.115	0.101	1.50
3	mSort + iRender + primeGen	0.227	0.225	0.20
4	mSort + iRender + primeGen + linSer	0.288	0.277	1.10
5	mSort + iRender + primeGen + linSer + polyMod	0.374	0.388	1.40
6	mSort + iRender + primeGen + linSer + polyMod + swk	0.411	0.421	1.00
7	2mSort + iRender + primeGen + linSer + polyMod + swk	0.495	0.5	0.50
8	2mSort + 2iRender + primeGen + linSer + polyMod + swk	0.542	0.515	2.70
9	2mSort + 2iRender + 2primeGen + linSer + polyMod + swk	0.616	0.625	0.90
10	2mSort + 2iRender + 2primeGen + 2linSer + polyMod + swk	0.648	0.668	2.00
11	2mSort + 2iRender + 2primeGen + 2linSer + 2polyMod + swk	0.713	0.759	4.60
12	2mSort + 2iRender + 2primeGen + 2linSer + 2polyMod + 2swk	0.766	0.788	2.20
13	3mSort + 2iRender + 2primeGen + 2linSer + 2polyMod + 2swk	0.890	0.835	5.53
14	3mSort + 3iRender + 2primeGen + 2linSer + 2polyMod + 2swk	0.872	0.860	1.20
15	3mSort + 3iRender + 3primeGen + 2linSer + 2polyMod + 2swk	0.919	0.916	0.30
16	3mSort + 3iRender + 3primeGen + 3linSer + 2polyMod + 2swk	0.987	0.933	5.40
17	3mSort + 3iRender + 3primeGen + 3linSer + 3polyMod + 2swk	0.988	0.983	0.50
18	3mSort + 3iRender + 3primeGen + 3linSer + 3polyMod + 3swk	0.982	0.99	0.80
19	4mSort + 3iRender + 3primeGen + 3linSer + 3polyMod + 3swk	0.956	1.0	4.70
20	4mSort + 4iRender + 3primeGen + 3linSer + 3polyMod + 3swk	0.991	1.0	0.90

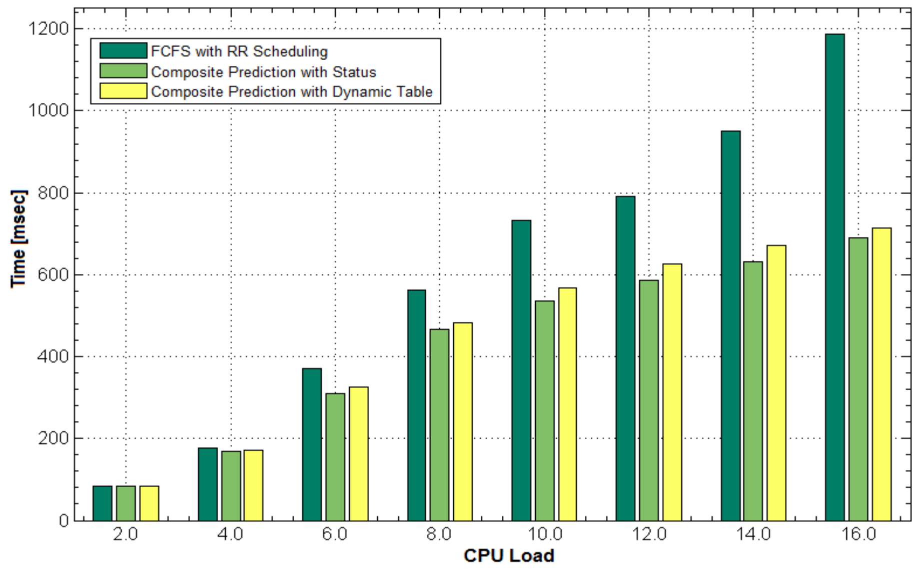


Figure 6.9: Task response time comparison among RR, TASM, and TASE models while the aggregate CPU load, L, was varied from $2.0 - 16.0$.

tasks, the prediction error incurd by the model is always below 6.0% in the dynamic changing environment.

6.4.4 Task Distribution Model Case Studies

The purpose of this section is to empirically measure the accuracy of the prediction based task assignment model in distributed systems with multi-core processors. A complete task assignment framework for RR scheduling with FCFS queue, TASM, and TASE have been implemented for measuring the task assignment efficiency. Real-world benchmark test programs shown in Table 6.1 with various CPU and memory requirements are selected as a group and used for the empirical study. Tasks features are also summarized in the Table 6.1.

Three sets of empirical studies have been conducted with these benchmark programs to compare the task assignment and execution time among the three studied models. Different scenarios with different sets of benchmark test loads have been

Table 6.3: Measured execution time, dispatch time, and total execution time of tasks with CPU load (L) ranging from 2.0 to 16.0 using RR with FCFS queue, TASM, and TASE models.

L	**TORQUE Model**			**TASM Model**			**TASE Model**		
	Exe. Time	**Disp. Time**	**Total Time**	**Exe. Time**	**Disp. Time**	**Total Time**	**Exe. Time**	**Disp. Time**	**Total Time**
2.0	83.99	0.32	84.31	82.76	1.02	83.78	83.57	0.34	83.91
4.0	176.30	0.56	176.86	168.76	1.79	170.55	171.62	0.60	172.22
6.0	369.89	0.88	370.77	310.72	2.81	313.53	322.17	0.94	323.11
8.0	563.68	1.20	564.88	467.92	3.83	471.75	479.48	1.28	480.76
10.0	731.76	1.44	733.20	536.23	4.59	540.82	562.31	1.53	563.84
12.0	792.31	1.76	794.07	586.76	5.61	592.37	623.79	1.87	625.66
14.0	951.63	2.08	953.71	632.21	6.63	638.84	668.23	2.21	670.44
16.0	1187.1	2.32	1189.45	689.24	7.40	696.64	713.47	2.47	715.94

prepared and carried out. The task assignment and execution statistics are stored in various files for statistical analysis. These obtained results are used for plotting graphs and deriving concluding remarks. Use of these benchmark programs enabled us to capture the behavior of real world applications for measuring the reliability and accuracy of task assignment models.

Figure 6.9 shows measured *task execution time* (time elapsed between dispatch to the time when execution is complete) bar graph for three studied models namely RR, TASM, and TASE. In this figure, the horizontal axis represents CPU load for the deployed benchmark tasks and the vertical axis represents task execution time in millisecond. Eight sets of empirical studies have been conducted in which CPU load, L, has been varied from 2.0 to 16.0. Tasks are selected from the table based on CPU requirements to match the required aggregate CPU load. The same group of benchmark programs containing identical CPU loads are considered for all 3 studied models to compare the task execution time among the models.

It can be observed from Figure 6.9 that when the aggregate CPU load for tasks are relatively small (i.e., $2.0 - 4.0$), the variation in task execution time is relatively

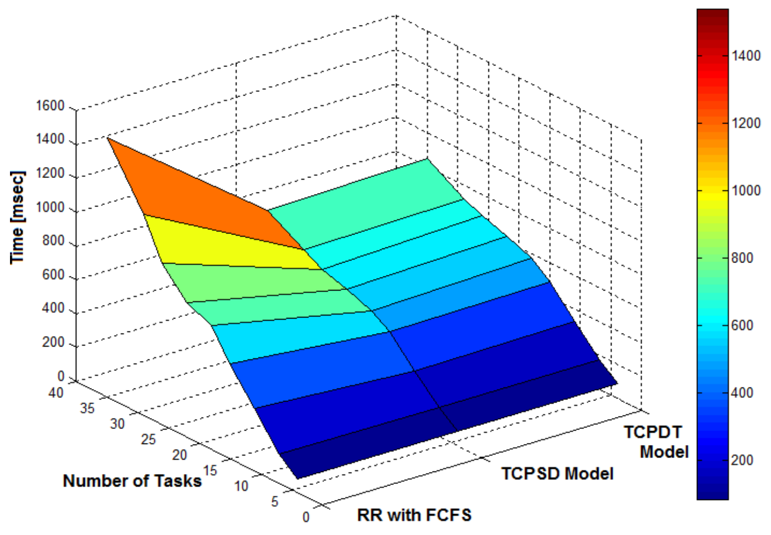


Figure 6.10: Total task execution time surface for RR, TASM, and TASE models while the task queue size, Φ, was varied from $4 - 40$ with benchmark tasks.

low. The RR model and prediction models perform about the same. As the aggregate CPU load increases for the group of tasks, both introduced prediction models clearly out performs the standard RR model. The composite prediction model intelligently assigns tasks to the most efficient node for execution based on task and node properties which has resulted in improved task execution time compared with the standard RR model. It can be further observed that between two prediction models, TASM model does marginally better than TASE model. Recall that the TASE model uses the new CPU availability model, Eq. 6.1, to predict available CPU for the assigned node where as the TASM model fetches the current status information. The TASM approach provides precise input to the models but it is expensive for clusters having a large number of compute nodes. The approach of TASE helps to save network bandwidth by transmitting fewer data packets for resource status collection (only once in the beginning) but a little slower because of the prediction error by the CPU usage prediction model while dynamically measuring and maintaining the status information table.

In further reporting the results of the studies, summary data are placed in Table 6.3 which shows the task execution time, task dispatch time, and total execution time in seconds for all 3 studied methods. To compare dispatch times among all 3 models, it can be observed that the task dispatch time of RR and TASE are very similar. The maximum difference between these two models is 150 msec when the value of $n = 40$ and $L = 16.0$. Task dispatch time for TASM model is the most because for assigning n tasks from the task queue, this model requires to fetch nodes' status n times from the grid. Though, TASM model inures the most task dispatch time among other models, it requires the minimum total task execution time for all set of empirical studies compared to other models. The execution time for TASM model is the best because the model always queries to have the most current and precise status data of nodes which helps the model to accurately pick the best suited node for the task.

As an end note, Figure 6.10 shows a comparison of total task execution time among three studied models, RR, TASM, and TASE, in various scenarios. In this figure, two horizontal axes represents the total number of tasks in the task queue, and the 3 studied models. The vertical axis represents the execution time in millisecond. This surface plot incorporates the scenarios in which various number of tasks in the task queue, ($4 \leq \Phi \leq 40$), to depict the effect of total task execution time. It can be observed for the empirical results that both TASM and TASE models perform better than the standard RR scheduling model for task assignment. The RR model degrades even more when there are I/O bound tasks in the queue. As the number of tasks increases in the queue, the RR scheduling fails to select proper nodes for assigning tasks which results in increased total task execution time.

6.5 Task Look-ahead using TASE Model

The new CPU availability model enables the TASE model to look-ahead a set of K tasks and find the best possible arrangement which can provide the minimum run-time for the batch. It is illustrated in the Figure 6.3 that the TASE model can look-ahead a set of incoming tasks from the run-queue. The model requires task properties and initial resource availability of nodes which can be extracted from the dynamic table. The model is then used for measuring the new CPU availability for each arrangement before actually placing the tasks into the run-queue. The measured resource availability are then compared to select the best arrangement for the set of tasks and dispatched for execution.

Algorithm 7 Exeter permutation algorithm which provides close to sorted order permutation of tasks based on *task id*.

```
Input: Task queue ($\Phi$), tid, start, size of $K$.
Subroutine PermuteTasks with arguments $\Phi$, start, $K$
  If $start == K - 1$ then
    Get task permutation combination.
    Measure the CPU usage with model Eqs. 6.1, 6.2, and 6.3.
    Assign the measured CPU in collections for task dispatcher.
  else
    for $i \leftarrow start \ ... \ K$ do
      Assign the $i^{th}$ task to temp container.
      Swap $i^{th}$ task with the start task.
      Recursively call the PermuteTasks with $\Phi$, start+1, and $K$
      Assign the $i^{th}$ task to start position
      Assign the task in temp container to $i^{th}$ position.
    end for
    Measure the actual CPU usage.
  end if
Collect the model and actual CPU usage data.
Store measured data to respective files.
Output: Model's Predicted Usage for the best permutation.
```

The Algorithm 7 provides the steps required for permuting K number of tasks from the task-queue. For example, if $K = 4$, then those 4 tasks can be arranged in $4! = 24$ different ways. Tasks are identified using their *id* number. For each arrangement, tasks are dispatched into the distributed nodes using RR policy and total task execution time is measured. The total execution time for each arrangement is then stored into a file for comparing with the TASE model.

6.5.1 Case Study Results

The major objective of this empirical work is to verify whether the task permutation has any effect to the task execution time. Five sets of test cases are conducted to empirically verify the task arrangement effect towards the run-time of tasks. In each case, 4 tasks are popped-out from the task queue and for each 24 arrangements (Algorithm 7 is used for task permutation), tasks are spawned into the compute node. Figure 6.11 shows the run-time of all 24 arrangements of the first 4 tasks superimposed with the run-time using TASE model. In this figure, the horizontal axis represents the run number (24 runs for 4 tasks) and the vertical column represents the total execution time in second. It can be observed from the figure that the total-run time is variable for different arrangements of tasks. The minimum and maximum run-times achieved are 521.18 *sec* and 767.35 *sec* respectively. Difference between the run-times is 246.17 *sec* which is 47.23% more than the minimum run-time.

The time difference is expected to increase even further when the compute nodes gets more tasks in their run-queue to process. The reason for time variation among different arrangements are because of the RR scheduling policy which fails to assign task to compute node depending on the resource availability. On the other hand, it can be observed from the Figure 6.11 that the total-time required by TASE model is 526.31*sec* which is very close to the best possible arrangement. The resource availability predictor of TASE model considers each core and hyper-threading instead of considering CPU as a whole for accurate prediction.

Table 6.4 consists of total execution time (ET) and percentage of additional time (δ_o) required by each arrangement for all test cases where $k = 4$. It can be observed from the table that the total run-time is variable based on different arrangements. The time taken by best possible arrangement among all test cases is 521.18*sec* which is around 1% better than the TASE model and the worst timing is 793.98*sec* which is 50.95% more than the TASE model. This extensive empirical

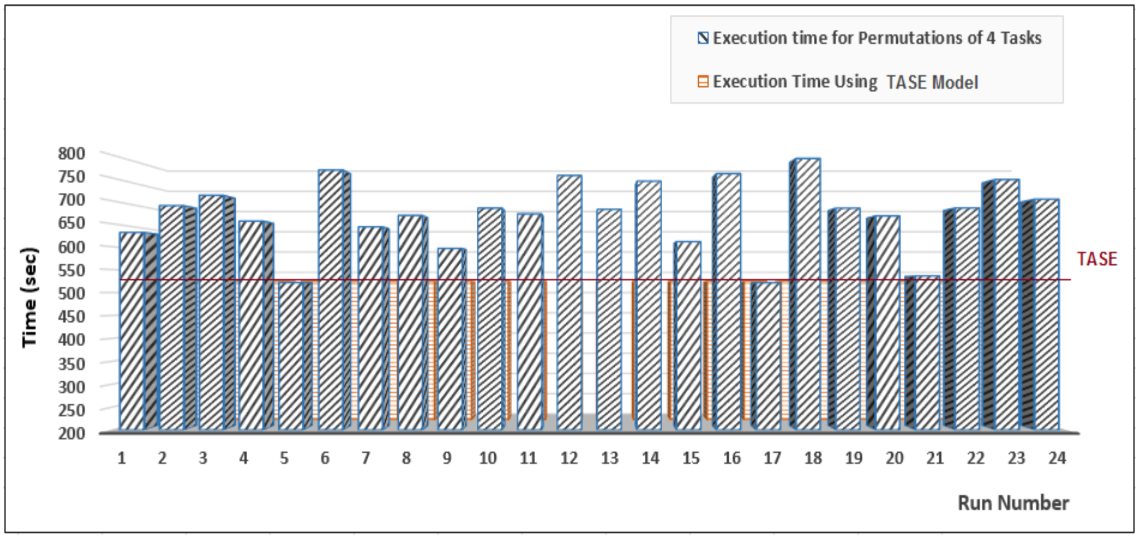


Figure 6.11: First k (here $k = 4$) tasks are taken from the task queue and dispatched for all $4! = 24$ permutations. For each permutation, tasks are dispatched using RR scheduling policy.

work and results validates the use of new CPU availability prediction model in TASE model to look-ahead tasks from the queue to minimize the run-time for a batch of tasks. Our extensive empirical evaluations have shown that TASE performs equally well (in terms of total completion time) in comparison with the best arrangement and TASM model as well, thereby avoiding the overhead of querying the nodes for their status each time a task is to be assigned.

6.6 Summary

This chapter has introduced two task assignment models for real-time scheduling of tasks in heterogeneous distributed nodes using composite prediction model. The composite prediction model proposed in Chapter 5 is derived from the CPU availability and memory models. Additionally, a new CPU usage prediction model is introduced to measure the usage of CPU prior to assignment of a task in to the run-queue. This introduce model is validated using wide variety of real-world benchmark

Table 6.4: Case study results for 24 permutations of 4 tasks in the run-queue. Measured execution time (ET) and Percentage of additional time, (δ_o), required compared with the TASE model.

Runs	**Case Study** 1		**Case Study** 2		**Case Study** 3		**Case Study** 4		**Case Study** 5	
	ET	δ_o	ET	δ_o	ET	δ_o	ET	δ_o	ET	δ_o
1	631.14	19.99	677.54	28.81	618.59	17.60	731.23	39.02	701.43	33.35
2	689.49	31.08	783.40	48.94	663.87	26.21	527.42	-0.57	702.30	33.52
3	711.97	35.36	764.05	45.26	599.57	13.99	679.07	29.10	693.13	31.77
4	655.70	24.66	729.00	38.59	652.73	24.09	789.75	50.14	593.87	12.90
5	521.72	-0.81	781.39	48.55	640.76	21.82	776.27	47.58	526.16	0.03
6	767.35	45.85	571.08	8.57	620.29	17.93	627.48	19.29	561.05	6.66
7	643.21	22.28	793.98	50.95	768.27	46.06	637.25	21.15	743.13	41.28
8	668.20	27.03	669.88	27.35	605.41	15.10	612.80	16.50	523.05	1.15
9	596.21	13.35	710.82	35.14	559.41	6.35	538.11	2.30	700.33	33.14
10	684.21	30.08	536.22	1.94	793.76	50.90	537.43	2.17	527.22	0.23
11	671.52	27.67	521.20	-0.91	589.53	12.08	555.41	5.59	594.29	12.98
12	755.22	43.58	690.46	31.27	680.58	29.39	722.72	37.40	768.29	46.06
13	681.43	29.55	583.70	10.97	760.50	44.58	534.85	1.68	572.58	8.86
14	742.82	41.22	774.80	47.30	736.83	40.08	694.02	31.94	546.51	3.90
15	611.14	16.19	679.52	29.19	583.20	10.87	784.28	49.10	651.77	23.91
16	759.37	44.37	567.03	7.80	763.84	45.22	676.88	28.68	728.04	38.41
17	521.18	-0.92	600.90	14.24	598.05	13.70	631.17	20.00	717.28	36.36
18	792.47	50.66	529.31	0.63	716.41	36.20	650.66	23.70	590.79	12.32
19	683.91	30.02	775.09	47.36	704.02	33.84	612.54	16.45	781.81	48.63
20	667.13	26.83	678.08	28.91	582.52	10.74	718.06	36.51	525.52	-0.19
21	536.07	1.91	730.37	38.85	660.30	25.53	708.72	34.74	698.83	32.86
22	684.37	30.11	649.81	23.54	764.54	45.35	707.29	34.47	621.92	18.24
23	746.46	41.91	528.42	0.46	557.71	6.03	576.94	9.68	532.67	1.27
24	703.57	33.76	531.44	1.03	686.07	30.43	666.66	26.74	746.40	41.90

programs and used in proposed TASE model. The model recalculates the CPU usage data for compute nodes in the prediction table after each task dispatch. Instead of a single CPU load value for the whole machine, the model considers CPU load of each core along with hyper-threading to accurately measure the available CPU for a new task.

A relative comparison among TORQUE resource manager, which uses Round Robin scheduling with FCFS queue, TASM, and TASE have been carried out in this chapter to demonstrate the prediction accuracy of the proposed models. An extensive set of empirical studies have been conducted in a distributed environment with heterogeneous compute nodes with real-world benchmark programs. The bar graphs and surface plots provides a clear visualization of measured execution efficiency and task completion time. It can be observed from the surface plots and bar graphs that the introduced TASM and TASE models clearly outperforms the TORQUE scheduler. The new CPU availability model used in TASE model provides accurate predicted CPU availability for incoming tasks to make intelligent task dispatch decision to distributed nodes.

Chapter 7

GPU Prediction Model for Compute-intensive Tasks

Using Graphics Processing Units (GPUs) to solve general purpose problems has received significant attention both in academia and industry. The readily available commodity hardware with an enormous computational potential has expanded the GPUs from only graphics rendering tool to powerful multi-core devices for broader applications. Efficiently utilizing the power of these devices, however, requires significant knowledge of the underlying architecture and the programming model. Hence, using the relevant features of the GPUs and selecting the proper parameters to extract the maximum performance is significant from the developers perspective. Predicting the performance of programs to be executed on GPU devices is therefore of utmost importance; also providing guidelines to programmers to tailor the code to achieve optimal results is equivalently important. In this chapter, we develop analytical model to predict the performance of GPUs for computationally intensive tasks. Our model is based on varying the relevant parameters - including the total number of threads, number of blocks, and number of streaming multi-processors - and predicting the performance of a program for a specified instance of these parameters. The model is tested using benchmark programs from the Nvidia CUDA GPU Computing SDK and also on multiple platforms including both Tesla and the latest Kepler GPUs; the performance prediction analysis is within an error bound of

0.13-5.69%. In addition, our model provide alternative feasible combinations of the parameters for efficient usage of the device and obtaining optimal performance. The approach can be used in the context of heterogeneous environments where distinct types of GPU devices with different hardware configurations are employed.

7.1 Introduction

GPU have establish their usefulness for solving general purpose large problems that requires massive parallel execution. The availability of GPUs as commodity hardware and co-processors to CPUs, and the relative low cost-to-performance ratio has propelled these devices to the forefront or research in both academia and industry. Compute Unified Device Architecture (CUDA), an extension to the C programming language, allows general-purpose problems to be solved using Nvidia GPUs. However, this paradigm shift has left developers striving for better performance from the available hardware. The general trend has been focusing on transferring compute intensive portions of tasks to the GPUs and exploiting parallelism in the existing code. Therefore, most of the research and studies relevant to GPUs focus on transferring the data from the CPU to the GPU and back, designing efficient data structures for the GPU, and utilizing available primitives to decrease memory latency [58] [66]. However, in reality, to exploit the full potential of these devices, understanding the underlying architecture and the basics of the programming model are required.

To illustrate, assume the total number of threads for a specific program is fixed at 512 threads. Let's further assume that the kernel has no divergence [1] and each thread executes 10 instructions. A developer has several arrangement options for the number and size of blocks to run the those 512 threads. Consider two arrangement cases: (a) 8 blocks with 64 threads in each block or (b) 64 blocks with 8 threads

[1]Thread Divergence: if a kernel contains divergence (multiple execution paths), different set of threads within the same warp will follow different control paths. In that situation, executing a warp requires several passes; one for each control path leading to variable execution time.

in each block. There is a significant difference between choice (a) and (b) though in both cases GPU will execute 512 threads. For the first case, the GPU needs to execute 8 blocks with 64 threads = 16 warps $\times$ 10 instructions = 160 instructions[2]. For the second case, 64 blocks with 8 threads = 64 warps $\times$ 10 instructions = 640 instructions which is *four times* higher compare to the first case. Thus, although the apparent the expected performance of both choices might be expected to be similar; in reality due to architectural considerations, execution times are widely different.

To be able to harness full potential of GPUs, extensive knowledge of the working environment is necessary, which makes utilizing the device to its maximum capacity a steep learning curve. Therefore, in this chapter, we focus our work on optimally using the device at hand. We concentrate on deriving analytical model that can predict the *execution efficiency* of GPU programs, i.e., GPU *kernels* based on certain parameters like the total number of resident blocks, number of threads in each block, the total number of blocks spawned in the device, the total number of streaming multi-processors available in the device, and others. The importance of our model is in the fact that it will help developers to unleash the full potential of the available hardware by predicting execution efficiency of thread blocks before placing them in the GPU run queue and also by providing suggestions to improve the efficiency by making arrangements for optimal execution time.

The benchmark programs used for empirical analysis to demonstrate the reliability and accuracy of the proposed model are professionally developed programs from the Nvidia CUDA GPU Computing SDK [73] . We validated the model using six different benchmark programs as follows: a) Monte Carlo Estimation of π b) Sobol Quasirandom Generator c) Super prime generator d) Sum of k Binomial Series e) Stirling's approximation of factorial, and f) CUDA Fast Fourier Transformation (FFT). The proposed analytical model is for GPU architectures in general.

[2]The block of threads is actually divided into sub-blocks called warps. Each SM splits its own blocks into warps and executes in single instruction multiple threads (SIMT) method. Currently, the size of each warp is 32 threads.

The benchmark programs are tested in both Tesla C1060 and Kepler 20 devices. The introduced model achieves very low prediction error with an error bound of 0.13-5.69%.

The rest of the chapter is organized in the following manner. In Section 7.2, we present a brief discussion of GPUs along with previous work related to performance prediction and analysis of GPUs. Section 7.3 incorporates the description of our analytical model and different parameters that are taken into consideration. Section 7.4 discusses the different benchmark programs that are used to evaluate the prediction model. Brief overview of the GPU architecture and the nuances of the CUDA programming model are studied in Section 7.4.2. Results of the prediction performance are shown in Section 7.4.5. This covers the case studies for all six different benchmark programs and their actual run time statistics on both Tesla C1060 and Kepler 20 devices. Finally, the summary of outcome of this chapter is discussed in Section 7.5.

7.2 Related Work

The CUDA environment allows programmers to create highly parallel applications by harnessing the power of GPUs [61]. These parallel applications can perform significantly faster than an equivalent program that is written to run on a standard CPU alone. The Nvidia CUDA Toolkit [73] provides several APIs for integrating a CUDA program into C and C++ applications. CUDA supports a heterogeneous programming environment where parts of the application code is written for the CPU and other parts of the application are written to execute on the GPU. The application is compiled into a single executable that can run on both devices simultaneously.

A variety of work has been conducted to analyze and optimize GPU programs for improving the performance of program execution compared to the amount of work for analyzing GPU architectures and provide reliable efficiency model. An analytical performance prediction for evaluating GPGPU (Genral Purspose GPU) applications [62] has been conducted to provide performance information to an auto-tuning compiler. Their predicted information assists the compiler to possible choices to more promising implementations. In this work, CUDA kernels have been analyzed and generated the corresponding string model. Their introduced string model is a concise representation of the operations of the kernel and summarizes how the kernel exercise major GPU micro-architecture features.

A quantitative analysis model for GPU architectures [65] has been carried out by developing a micro-benchmark based performance model for Nvidia GeForce 200-series GPUs. The model proposed by [65] identifies GPU programs bottlenecks and quantitatively analyzes performances which may allow programmers to predict the benefits of potential program optimizations. They have also conducted empirical studies to improve the performance of the GPU based on performance analysis models focusing on the instruction pipeline and utilizing the shared and global memory efficiently.

Another quantitative analysis of performance predictions for general purpose computation on GPUs [64] has been conducted and factors that determine GPU performance has been identified. The factors were then classified into three categories: data-linear, data-constant, and computation-dependent. Based on the characteristics of these factors, a performance model has been proposed for each factor. For predicting the performance of bio-sequence, these models were utilized and verified[3].

A CPU availability and memory model has been introduced in [17] for predicting (and measuring) the overall nodes' efficiency for concurrent thread execution on a time-shared system. A composite prediction has been derived using the CPU availability and memory model. Introduced models are empirically derived upper- and lower efficiency bounds. Experiments have shown that the introduced models can bind the thread execution efficiency values very well and bounds are significantly tight. The execution efficiency of a batch of tasks can be predicted before placing the

[3]But, most of the research and studies relevant to GPUs paradigm focus on feeding the data and programs on the device and gain some performance in terms of through-put.

tasks into the run-queue. Two intelligent task assignment models are introduced for dynamic scheduling of tasks in heterogeneous cluster using the composite prediction model. The chapter has predicted an assignment of tasks to compute nodes such that the total time taken to execute all the tasks is minimized.

Research has also been performed by creating performance models based on the data transfer between the CPU and a GPU [67]. Although this can be significant, for many compute intensive tasks, there is a minimal amount of data actually being transferred between the CPU and the device. Thus, other models for analysis and performance benefits are required. There has been work done on improving the performance of CUDA kernels by restructuring loops using loop tiling and coalesced memory access. However, this approach requires significant changes to the programs themselves and would incur added complexity to the code. There has also been research focusing on using all available cores with high degree of multi-threading and also taking advantage of various available primitives like memory coalescing [57, 59]. Previous study on framework for performance analysis have determined bottlenecks present in the code and methods to remove them [53], and provides improvement suggestions for various optimization techniques.

7.3 Analytical Model

The primary focus of this section is to derive a prediction model for estimating the efficiency of thread execution independent of GPU architectures. It is necessary to have accurate model for estimating the execution efficiency because of the dynamic nature of GPU systems and their workload. As new thread blocks are assigned or existing blocks complete execution, the efficiency of a given GPU system can change in a short interval of time. Therefore, existence of prediction models are important because there exists a wide range of applications and scientific models (e.g., graphical, meteorological, economical, and others) that requires extensive use of both GPU and CPU resources, repeatedly. Assigning a set of thread blocks can

Table 7.1: Terms and definitions of GPU efficiency prediction model parameters.

Terms	Definition
ρ_{sm}	Number of Streaming Multi-core processors (SM) in the device. The number of multi-processors varies from architecture to architecture.
ρ_w	Maximum number of resident warps in a streaming multi-core processor.
ρ_b	Maximum number of resident blocks in a streaming multi-core processor. It is the number of blocks that a SM can process simultaneously.
ρ_t	Maximum number of resident threads in a streaming multi-core processor. That is, the maximum number of threads that an SM can track and schedule simultaneously.
σ_w	Size of warp. Size for both Tesla C1060 and Kepler K20 is 32.
σ_b	Maximum size of a block. That is, the maximum number of threads in a block. For Tesla C1060, $\sigma_b = 512$ and for Tesla K20, $\sigma_b = 1024$.
β	Number of blocks spawned in the GPU device. The value of β will be same for different ϑ while running different benchmark programs.
ϑ	Number of threads per block (block size). The value of ϑ will be different for same β while running different test cases.
τ	Ideal thread execution time. The value will be different for different benchmark programs.

utilize the prediction model to determine the best arrangement of blocks and threads in each block that can minimize the total execution time prior to the placement of kernel in the GPU run queue.

An analytical framework for estimating the overall execution efficiency for batches of thread blocks is derived for GPU systems. Table 7.1 contains the notation and definitions of required parameters of the model. The introduced prediction model incorporates the following three major observed categories while GPU executes several blocks of threads.

- When the number of blocks to be executed is more than the aggregate number of resident blocks for the GPU.
- When the number of warps to be executed is more than the aggregate number of resident warps for the GPU.

- When the total number of threads to be executed is more than the aggregate number of resident threads for the GPU.

Effects specified above can change the kernel execution time significantly and needs to be incorporated to derive an accurate prediction model. The execution time is ideal, denoted by τ, when the number of blocks and threads in each block is less than the aggregate resident blocks and threads for the GPU (this number varies by GPU architecture). The following model incorporates the effect for the case when the number of thread blocks spawned for execution is more than the aggregate resident blocks for all multi-core processors of the GPU. Input parameters to the following model includes: the number of thread blocks spawned in the GPU (β), maximum number of resident blocks per streaming multi-core processor (ρ_b), and the number of streaming multi-core processor in the device (ρ_{sm}) for measuring the execution time (ξ_{rb}).

$$\xi_{rb} = \begin{cases} \tau & \text{if } \beta \leq (\rho_b \times \rho_{sm}), \\ \text{Max}\left(1, \left\lceil \frac{\beta}{\rho_b \times \rho_{sm}} \right\rceil\right) \times \tau & \text{else} \end{cases} \tag{7.1}$$

When the number of thread blocks spawned is more than the number of aggregate resident blocks (i.e., $\beta > (\rho_b \times \rho_{sm})$) that can be scheduled in one cycle, the execution time increases by τ for the next cycle execution. The execution time increases by a factor of τ if a new cycle is required. Figure 7.1 shows a graph for the model in which the horizontal axis represents the number of thread blocks spawned for execution in the GPU and the vertical axis represents the total kernel execution time in milliseconds. The model is executed with $\rho_{sm} = 13$, $\rho_b = 16$, $\tau = 14.5$, and the β is increased from 16 to 1440. It can be observed from the figure that for the first 13 cases when $\beta \leq (\rho_b \times \rho_{sm})$, the execution time is same as the value of τ. When the value of β is more than $(\rho_b \times \rho_{sm})$, the execution time increases by the value of τ for the next scheduling cycle. Similarly, for each new cycle, the value of τ is added to reflect the observed scheduling behavior.

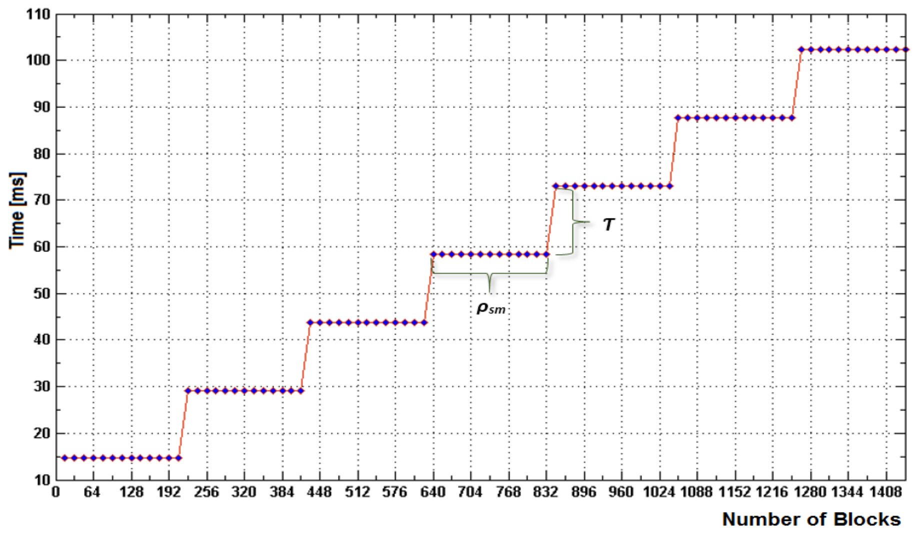


Figure 7.1: Execution time is proportional to the number of blocks (β). The number of threads in a block (ϑ) is fixed at 32 while β is increased from 16 to 1440.

As described earlier, the block of threads assigned to a SM are actually divided into sub-blocks called *warps*. Each SM splits its own blocks into warps (currently with a maximum size of 32 threads) [91]. Instructions are issued per warp and execute in a Single Instruction Multiple Thread (SIMT) fashion by a CUDA multiprocessor. The first warp (0) within a resident block contains threads with id $0, ..., 31$; the second warp (1) contains threads $32, ..., 63$; the third warp (2) contains threads $64, ..., 95$; etc. The relation between the thread index and warp index for one-dimensional blocks is $32 \times i, ..., 32 \times (i + 1) - 1$ where i is the warp index. The size of warps can be less than 32 active threads for one of the following reasons.

1. The number of threads per block is not divisible by 32,

2. The active kernel contains one or many divergent blocks and threads that did not take the current path are marked idle, or

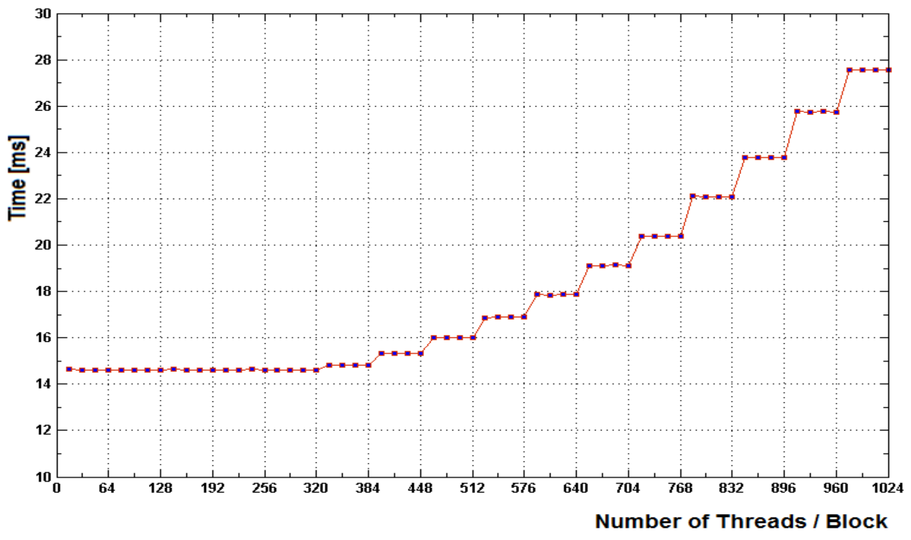


Figure 7.2: Execution time is proportional to the number of resident warps (ρ_w). The number of blocks (β) is fixed at 16 while the number of threads in a block (ϑ) is increased from 16 to 1024.

3. A thread in the warp has finished it's work and terminated. Even if a warp holds less than 32 threads, in most situations, it will be executed similarly as if it contains 32 threads.

The number of resident warps varies depending on the specific GPU architecture. When more resident thread blocks are placed into a SM, the warp occupancy increases which leads to higher kernel execution time. The warp occupancy is defined as the *number of active warps* divided by the *maximum active warps* in a SM [88]. For harnessing the power of GPU, increased warp occupancy is desired that nearly eliminates the latency due to context switching. Most of the GPU device has built in hardware scheduler that makes context switching fast among warps. A higher occupancy reduces processor idle time (SM may stall due to unavailability of data or busy functional units) and improves overall performance [90]. Input parameters to the model for deriving the estimated execution time when the number of threads increases in a block, denoted by ξ_{rw}, includes: the number of thread blocks spawned in the GPU (β), number of threads in each block (ϑ), maximum number of resident warp in each SM (ρ_w), and the size of each warp (σ_w).

$$\xi_{rw} = \left\{ \frac{\beta \times \vartheta}{(\rho_w \times \sigma_w \times \rho_{sm})} \right\} \times \tau \tag{7.2}$$

Eq. 7.2 holds when $(\beta \times \vartheta) \leq (\rho_w \times \sigma_w \times \rho_{sm})$. When the number of warps to be executed is more than the aggregate number of resident warps for the GPU, additional warp scheduling overhead is added with the ideal thread execution time. Figure 7.2 depicts the effect of resident warp in which the horizontal axis represents the number of threads per block and the vertical axis represents the total execution time in milliseconds. The model is executed with $\beta = 16$, $\rho_{sm} = 13$, $\rho_w = 64$, $\sigma_w = 32$, $\tau = 14.5$, and the ϑ is increased from 16 to 1024. The curve provides a clear visualization of the effect of concurrent warp execution in a single SM. As the number of warps in a SM increases to ρ_w, the execution time can increase up to $(2 \times \tau)$ when it reaches the occupancy. Eq. 7.2 explains the situation only when $(\beta \times \vartheta) \leq (\rho_w \times \sigma_w \times \rho_{sm})$.

When the total number of threads $(\beta \times \vartheta)$ to be executed is more than the aggregate number of resident threads for the GPU, the execution time can be explained by the following series:

$$\tau \text{ ... } 2\tau \quad 3\tau \text{ ... } 4\tau \quad 5\tau \text{ ... } 6\tau \quad \text{...} \quad \xi_{rw} \text{ ... } (\xi_{rw} + 1).$$

When both the number of blocks and threads in a block increases, the situation gets complex and requires both Eqs. 7.1 and 7.2 to explain the situation. Because of the increased number of threads in a block, a SM cannot accommodate the maximum number of resident blocks. For example, though Nvidia K20 can have maximum 16 resident blocks, when $\vartheta = 1024$ (maximum number of threads in a block), it can place only 2 blocks in a SM because the maximum number of resident threads in a SM is 2048. Figure 7.3 shows the composite effect of increased blocks and threads in a block on total execution time of the GPU kernel. In this figure, two horizontal axes represents ϑ and β and the vertical axis represents the total execution time in milliseconds. With careful observation, it can be seen that for each case when $\beta > (\rho_b \times \rho_{sm})$, the execution time jumps by the value of τ. Similarly, when $(\beta \times \vartheta) > (\rho_w \times \sigma_w \times \rho_{sm})$, for each $(\rho_w \times \sigma_w \times \rho_{sm})$, the execution time increases utilizing Eq. 7.2. The total execution time increases sharply for large number of blocks and its sizes.

The composite prediction model which incorporates all major specified effects includes the input parameters: ideal thread execution time, execution efficiency when $\beta > (\rho_b \times \rho_{sm})$, and execution efficiency when threads in a SM $> (\rho_w \times \sigma_w)$. The model equation is derived from Eqs. 7.1 and 7.2 to reflect the observed effect in total thread execution time (κ).

$$\kappa = \begin{cases} \frac{\tau}{(\tau+\xi_{rw})}, & \text{when } (\beta \times \vartheta) \leq (\rho_w \times \sigma_w \times \rho_{sm}) \\ \frac{\tau}{(\xi_{rb}+\xi_{rw})}, & \text{when } (\beta \times \vartheta) > (\rho_w \times \sigma_w \times \rho_{sm}) \end{cases} \tag{7.3}$$

When the value of $(\beta \times \vartheta) \leq (\rho_w \times \sigma_w \times \rho_{sm})$, the τ is divided by an expression which incorporates the overhead of warp occupancy that is explained in the Figure 7.2. Next, when $(\beta \times \vartheta) > (\rho_w \times \sigma_w \times \rho_{sm})$, the τ is divided by an expression which contains the overhead of both $\beta > \rho_b$ and $\vartheta > \rho_t$ expressed in Eqs. 7.1 and 7.2.

Figure 7.4 depicts the execution efficiency surface measured by Eq. 7.3. In this figure, two horizontal axes represents ϑ and β and the vertical axis represents execution efficiency on a scale of 0.0 to 1.0. The efficiency surface diagram clearly visualizes the composite effect of increased number of β and ϑ. The model depicts a sharp performance degradation as soon as the $\beta > \rho_b$. The efficiency degrades sharply when the value of τ is added because of the requirement of a new scheduling cycle. Similar degradation of efficiency can also be observed when SM occupancy gets full and requires next execution cycle (shown in Figure 7.2).

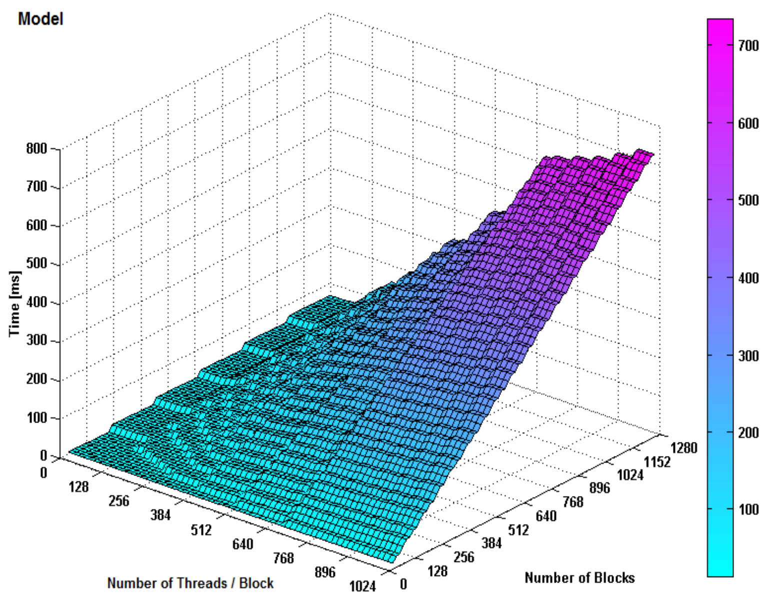


Figure 7.3: Estimated total execution time surface diagram using the introduced prediction model. Here, ϑ is increased from 32 to 1024, β is increased from 16 to 1248, and $\tau = 14.5$.

7.4 Empirical Studies

7.4.1 Overview

The major contribution of this study is to empirically measure the execution efficiency of kernels in GPUs as a function of aggregate blocks and threads in each block. Six separate test cases have been conducted and are presented here to validate and show the accuracy of the introduced prediction model. Table 7.2 shows the set of benchmark programs used for measuring the overall efficiency of GPU devices. The Monte carlo estimation of π, Sobol Quasirandom Generator, Super prime number generator, Sum of binomial series, Stirling's Approximation of Factorial, and CUDA FFT are professionally developed programs by Nvidia. These programs are available in Nvidia CUDA SDK and used as benchmark programs so that the prediction model can be verified with real-world applications in a highly dynamic environment.

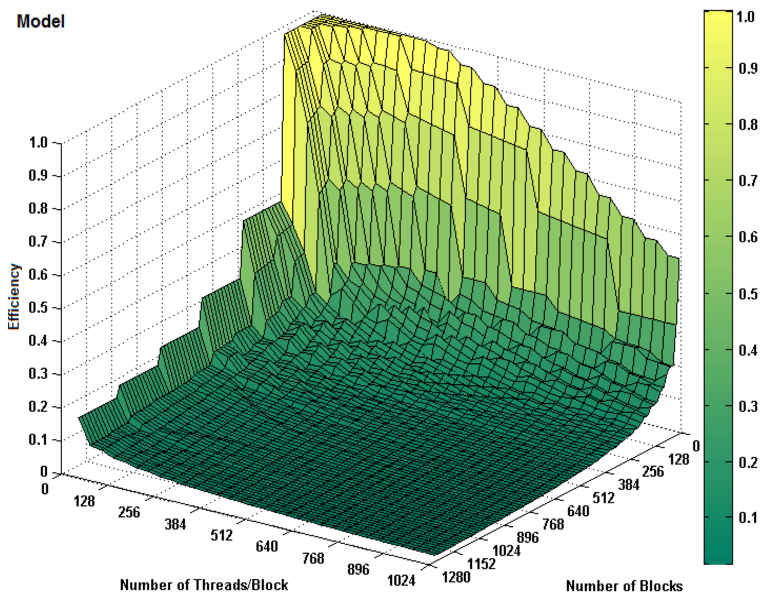


Figure 7.4: Estimated efficiency surface diagram using the introduced predicted model (associated with Eq. 7.3). Here, ϑ is increased from 32 to 1024, and β is increased from 16 to 1248, and $\tau = 14.5$.

Algorithm 8 provides the high-level pseudo-code for the empirical framework used to measure the execution time by deploying benchmark programs. The algorithm takes maximum number of blocks, maximum number of threads in a block, number of threads to increment in a block after each iteration, number of blocks to increment after each iteration, and the value of τ as input. Initially, 16 blocks are spawned in the GPU each with 32 threads (1 warp). The Nvidia CUDA *clock* [73] toolkit is used for accurately measuring the kernel execution time. The measured time is stored in a file and the value of blocks and threads are incremented using the input value. The program continues until it finishes sampling up to maximum number of blocks and threads in a block, as provided in the input.

Algorithm 8 is used for measuring execution times for all benchmark programs listed in Table 7.2. The user is also required to specify the target benchmark program along with the other parameters mentioned above. The value of current β and ϑ are

Algorithm 8 Spawning Nvidia benchmark kernels for measuring the execution efficiency.

```
Input: Benchmark program ID ($BID$), Maximum number of blocks ($bMax$), Maximum number of threads in a block ($tMax$), Block increment size ($incBlk$), Thread increment size ($incTrd$), and the ideal execution time for the benchmark program ($\tau$)
Output: Kernel execution report (execution time, launch time, total time) for variable block size and number of threads in a block.
Initialize required program variables with associated values
for bCount $\leftarrow incBlk\ ...\ bMax$ do
  for tCount $\leftarrow incTrd\ ...\ tMax$ do
    numThreads $\leftarrow bCount\ \times\ tCount$
    Cuda memory copy numThreads from Host to Device
    Start CUDA clock toolkit for measuring event start time
    Invoke CUDA benchmark kernel using $BID$, $bCount$, $tCount$, and other
      necessary parameters
    Synchronize benchmark threads (based on BID)
    Wait for threads to complete assigned work
    Stop CUDA clock toolkit for measuring the event end time
    Read the execution result and verify for accuracy
    Store the kernel execution report for bCount and tCount in a CSV file
    if $(\beta \times \vartheta) \leq (\rho_w \times \sigma_w \times \rho_{sm}))$ then
      $\kappa = \frac{\tau}{(\tau+\xi_{rw})}$
    else if $(\beta \times \vartheta) > (\rho_w \times \sigma_w \times \rho_{sm}))$ then
      $\kappa = \frac{\tau}{(\xi_{rb}+\xi_{rw})}$
    end if
    Store the $\kappa$ for associated bCount and tCount
    tCount $\leftarrow incTrd + tCount$
    Deallocate host and device memory used for collections
  end for
  bCount $\leftarrow incBlk + bCount$
end for
Close all input and output streams, pipes, files, and connections
```

used to calculate the number of threads spawned in the GPU. The value of execution efficiency (κ) is then measured using Eq. 7.3 and stored in CSV files along with other measured run-time value for later statistical analysis.

7.4.2 Environment

Two different GPU devices are used for evaluating the proposed efficiency prediction model for parallel thread execution. We have used Tesla Kepler 20 and Tesla C1060 GPU cards for these empirical studies. Two independent case studies are conducted using the same real-world benchmark programs for validating the accuracy of the model.

The first set of empirical studies are conducted using Tesla Kepler 20 device. The Tesla K20 card contains a large set of processor cores with the ability to directly address global memory. The Tesla K20 GPU card contains 13 streaming multi-processors, 192 cores in each streaming multi-processor (SMs) giving a total of 2496 cores with core clock 0.71 GHz. Tesla K20 uses CUDA API, Compute Capability 3.5, Global Memory 5 GB, Shared memory 48 KB, Memory bandwidth 208 GB/Sec, Number of 32-bit registers per streaming multi-processor 64K. This model can accommodate per streaming multi-processor maximum of 16 resident blocks, 1024 threads per block, 2048 resident threads, and 64 resident warps. The size of warp is 32 threads. The Tesla K20 card is hosted by an Intel(R) Xeon(R) quad-core machine with 2.60 GHz clock and 32GB of RAM.

The second set of empirical case studies are done in Tesla C1060 GPU device which is a previous generation card and contains fewer processor cores. The Tesla C1060 GPU card contains 30 streaming multi-processor, 8 cores in each streaming multi-processor, core clock 1.3 GHz, uses CUDA API, Compute Capability 1.3, Global Memory 4 GB, Shared memory 16 KB, Memory bandwidth 102 GB/Sec, Number of 32-bit registers per streaming multi-processor 16K. This model can accommodate per streaming multi-processor a maximum of 8 resident blocks, 512 threads per block, 1024 resident threads, 32 resident warps, and the size of warp is 32 threads. The Tesla C1060 card is hosted by an identical Intel(R) Xeon(R) quad-core machine with 2.60 GHz clock and 32GB of RAM.

7.4.3 Programming Model

CUDA provides a general-purpose parallel programming model. The programming language for CUDA extends C and defines C-like functions, called kernels, which are executed in parallel by different threads on the GPU. Threads are grouped together in blocks, which can be either one, two or three dimensional, and each thread within a block is identified by its unique *threadID*. All the blocks of threads, identified by *blockID*, form a one or two dimensional grid. Threads within a block can co-operate among themselves through the shared memory and synchronize their execution to coordinate memory accesses. Blocks of threads are assigned to the streaming multiprocessors by the CUDA scheduler. Data is transferred from the CPU to the GPU and back using memory copy functions.

Computation heavy tasks are transferred from the CPU to the GPU, and by using all the available processor cores and multi-threading options, problems with a large number of computations can be solved efficiently. Data required for computations is transferred to the GPU from the CPU using the global or device memory on the GPU; similarly computed results are transferred back from the GPU to the CPU using the same.

7.4.4 Benchmark Programs

Table 7.2 shows the benchmark programs used for validating the proposed prediction model. The benchmark programs are taken from Nvidia CUDA GPU SDK and re-framed to fit our empirical test environment. The reason for using these professionally developed programs by Nvidia as benchmark program in this chapter

Table 7.2: Benchmark programs used from Nvidia CUDA GPU SDK for validating the introduced prediction model.

Name	**Description**	**Genre**
Monte Carlo Estimation of π	This sample uses Monte Carlo simulation for Estimation of π and uses the Nvidia CURAND library.	Compute Intensive
Sobol Quasirandom Generator	This sample implements Sobol Quasirandom Sequence Generator.	Computational Finance
Prime Number Generator	Generates High Order Prime Numbers	Compute Intensive
Sum of Binomial Series	This program generates and adds first N binomial series and produces the aggregate sum.	Computational Finance
Stirling's Approximation Error	This sample API measures the approximate value of a large factorial number using Stirling's model and finds the approximation error.	Compute Intensive
Simple CUFFT	CUFFT is used to compute the 1D-convolution of some signal with some filter by transforming both into frequency domain, multiplying them together, and transforming the signal back to time domain.	Compute Intensive

is to demonstrate the accuracy and reliability of introduced prediction model while executing real-world applications.

The benchmark programs listed in Table 7.2 consists of expressions that are highly compute intensive and thus ideal for utilizing them in our empirical case studies. Nvidia CUDA provides a *clock API* to measure the performance of kernel accurately. We have utilized the CUDA clock toolkit in our empirical framework to measure the precise execution time of the benchmark programs.

7.4.5 Case Study Results

The major objective of this section is to measure the accuracy of the introduced prediction model for estimating the execution efficiency of compute intensive GPU kernels. The model is verified using both Tesla C1060 and Kepler 20 devices. All

Table 7.3: Empirical results using **Tesla Kepler 20** GPU device for the Sum of Binomial Series benchmark program.

β	**Number of Threads per Block (ϑ)**									
	32	**64**	**96**	**128**	**160**	**192**	**224**	**256**	**288**	**320**
16	14.60	14.61	14.62	14.63	14.62	14.62	14.62	14.62	14.63	14.64
64	14.61	14.60	14.63	14.62	15.42	15.91	16.90	17.73	21.34	22.57
112	14.60	14.53	15.20	16.69	21.38	24.35	27.93	42.31	42.74	41.72
160	14.54	15.19	17.84	22.47	40.29	40.58	44.35	45.28	52.16	65.71
208	14.55	15.83	20.80	27.64	43.00	45.12	52.69	55.23	70.21	77.90
224	29.02	30.36	35.30	42.16	42.97	46.65	56.08	69.69	75.50	78.63
272	29.05	30.37	35.47	43.40	50.60	66.41	70.39	79.28	84.12	95.45
320	29.07	30.54	36.56	45.43	67.23	71.02	80.39	98.46	102.9	117.3
368	29.08	31.60	39.95	51.77	69.03	78.60	96.55	103.4	123.5	127.1
416	29.02	31.63	41.58	55.38	75.08	93.54	102.2	110.4	129.7	147.5
432	43.45	46.13	56.03	69.73	78.60	94.69	105.3	124.9	134.9	147.9
480	43.50	46.15	56.19	70.99	93.14	99.24	124.6	134.3	153.7	170.8
528	43.51	46.32	58.16	74.42	94.91	117.8	132.6	152.3	164.8	178.4
576	43.50	46.75	60.69	79.36	101.6	120.7	139.1	161.9	181.4	199.4
624	43.51	47.41	62.39	83.10	105.8	130.2	152.4	165.5	192.9	209.2
640	57.87	61.91	76.75	97.30	118.9	131.1	156.4	180.1	203.4	222.6

benchmark programs listed in Table 7.2 are deployed in both C1060 and K20 devices and the efficiency are measured separately. For the Tesla C1060 device, the number of blocks are varied from 8 ($\rho_b = 8$ for C1060 device) to 1248 and threads in each block are increased from 16 to 512 ($\sigma_b = 512$ for C1060 device). For the Tesla K20 device, the number of blocks are varied from 16 ($\rho_b = 16$ for K20 device) to 1248 and threads in each block are increased from 32 to 1024 ($\sigma_b = 1024$ for K20 device). Measured run-time in Table 7.3 is in millisecond. The ideal run-time for the sum of binomial series in Kepler 20 GPU is 14.6 ms and in Tesla C1060 is 54.9 ms.

For collecting empirical data of every possible case, all benchmark programs are independently deployed 2496 times in which number of blocks and threads in a block are selected in increasing order (specified above). A snapshot of the empirical results is presented in Table 7.3 for the Sum of Binomial Series benchmark program. Table 7.3 contains partial data in which β is increased from 16 to 640 and ϑ is increased from 32 to 320 (i.e., from 1 warp to 10 warps per block). For the Tesla

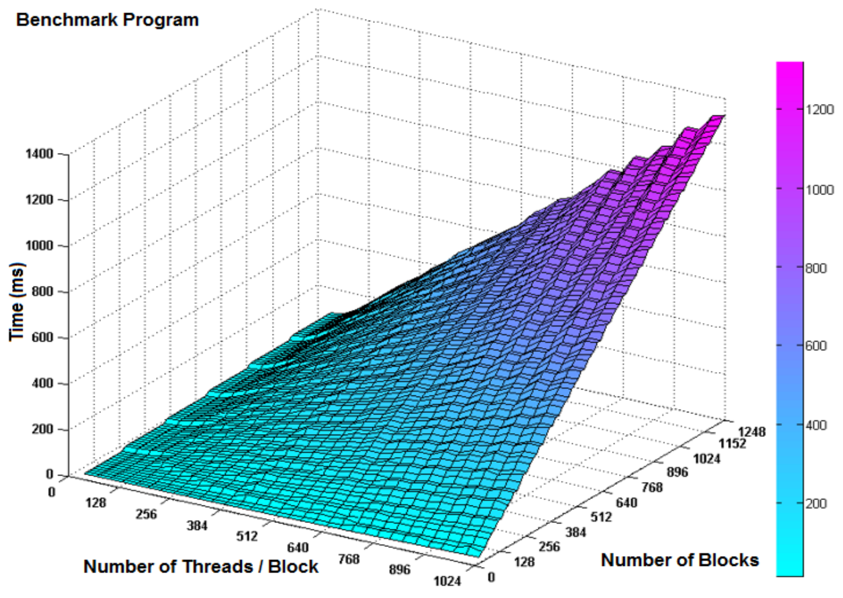
Table 7.4: Empirical results using **Tesla C1060** GPU device for the Sobol Quasirandom Generator benchmark program.

β	**Number of Threads per Block (ϑ)**									
	32	**64**	**96**	**128**	**160**	**192**	**224**	**256**	**288**	**320**
16	54.9	56.8	57.5	59.9	61.8	65.7	71.7	79.6	88.4	97.1
64	56.5	65.6	88.5	114.9	143.6	172.0	201.0	229.5	258.4	286.9
112	58.2	79.6	115.3	152.9	191.6	229.5	268.0	306.0	346.7	383.9
160	61.0	114.9	172.6	229.5	287.3	352.5	401.9	458.8	516.7	573.7
240	68.8	152.9	230.3	306.0	384.4	458.8	535.9	611.9	688.9	764.8
256	123.7	210.3	288.8	366.1	430.9	516.3	607.7	691.5	775.0	860.4
304	125.4	218.5	318.7	420.8	526.7	639.2	736.8	841.3	947.0	1052
352	126.7	232.5	345.5	458.8	574.6	688.6	803.9	917.8	1033	1147
400	129.9	267.8	402.9	535.4	671.6	803.0	937.8	1071	1205	1338
480	137.7	305.8	460.5	611.9	766.1	926.1	1072	1224	1380	1531
496	192.5	363.1	519.2	671.7	813.9	975.3	1143	1303	1464	1625
544	194.2	371.3	548.9	726.8	923.6	1090	1273	1453	1638	1818
592	195.3	385.3	575.7	764.8	958.9	1147	1340	1530	1722	1912
640	198.7	420.6	633.2	841.3	1053	1262	1474	1683	1896	2105
720	206.4	458.6	690.7	917.8	1149	1377	1608	1836	2066	2294
736	261.3	516.1	749.6	978.0	1211	1434	1679	1915	2155	2392

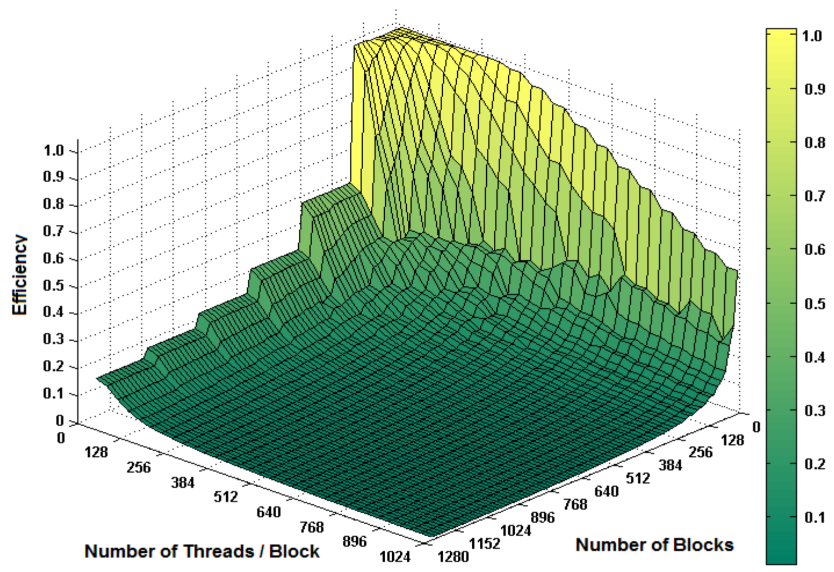
K20, there are 13 SMs; it can be observed from the table that for $\vartheta = 32$, when $\beta \leq (\rho_b \times \rho_{sm})$, in this case $\beta \leq (16 \times 13 = 208)$, the execution time is close to the value of τ. As soon as the value of $\beta > (\rho_b \times \rho_{sm})$, the execution time increases by the value of τ for the additional execution cycle. The same effect can be observed when $\beta = 432$, and 640. This behavior is modeled according to previously derived Eq. 7.1.

It can be further observed from Table 7.3 that for $\beta = 64$ and ϑ ranging from 32 to 320, as the $(\beta \times \vartheta) > \rho_t$, the execution time increases around $(2 \times \tau)$ depending on the number of warps. This behavior is modeled using Eq. 7.2. Next, when the total number of threads spawned $(\beta \times \vartheta)$ is more than the occupancy, $(\rho_w \times \sigma_w \times \rho_{sm} = 64 \times 32 \times 13 = 26,624$ for the Tesla K20 device), the specified series in Section 7.3 can be observed in the Table 7.3. For $\beta = 208$ and $\vartheta = 128$, the number of threads spawned in the device is $208 \times 128 = 26,624$ (i.e., maximum occupancy reached)[4].

[4]The maximum number of resident threads in a Kepler 20 SM is 2048 though a thread block can contain a maximum of 1024 threads.



(a)



(b)

Figure 7.5: Surface diagrams for the Sum of Binomial Series benchmark program (a) Surface diagram shows increasing run-time while ϑ increased from 32 to 1248 and β increased from 16 to 1024 (b) The measured efficiency surface diagram with respect to the ideal execution time (τ).

As soon as the ϑ or β increases, the execution time increases depending on the number of execution cycle. This observed behavior is modeled using Eq. 7.3.

Table 7.4 contains a snapshot of the empirical data from Tesla C1060 GPU for the Sobol Quasirandom Generator benchmark program. The value of β is increased from 16 to 736 and ϑ is increased from 32 to 320 (i.e., from 1 warp to 10 warps per block). In Tesla C1060 GPU device, there are 30 SM; it can be observed from the table that for $\vartheta = 32$, when $\beta > (\rho_b \times \rho_{sm})$, in this case ($8 \times 30 = 240$), the execution time increases to ($68.8 + 54.9 = 123.7$ ms) by the value of τ ($\tau = 54.9$ ms) for the additional execution cycle. The same effect can be observed when $\beta = 496$, and 720. Similar scheduling effects can be observed in Table 7.3 for $\beta = 240$ when the total number of threads spawned ($\beta \times \vartheta$) is more than the occupancy of the device, ($\rho_w \times \sigma_w \times \rho_{sm} = 32 \times 32 \times 30 = 30,720$ for the Tesla C1060 device).

Figure 7.5 (a) shows the complete measured run time surface for the Sum of binomial series benchmark program. In this figure, the two horizontal axes represents β and ϑ and the vertical axis represents the total run time in milliseconds. The surface depicts the complete 2496 independent test run results to capture all possible scenarios by varying the values of β and ϑ. It can be observed that each time when $\beta \times \vartheta$ crosses ($\rho_w \times \sigma_w \times \rho_{sm}$), the execution time increases for new scheduling cycles. Similarly, when β crosses ($\rho_b \times \rho_{sm}$), the execution time increases by τ. The run-time surface of sum of binomial series shown in Figure 7.5 (a) and the model surface shown in Figure 7.3 depicts similar shape and pattern.

Figure 7.5 (b) shows the measured efficiency surface (using Eq. 7.3) for the same test run data of sum of Binomial Series benchmark program. It can be observed from Figure 7.5 (b) that the efficiency value decreases significantly when β and ϑ increases beyond the capacity of scheduling cycle. The cumulative effect of Figure 7.1 and Figure 7.2 can be observed in the diagram. This surface diagram clearly visualizes the effect of increased β and ϑ by including 2496 test case results. The efficiency

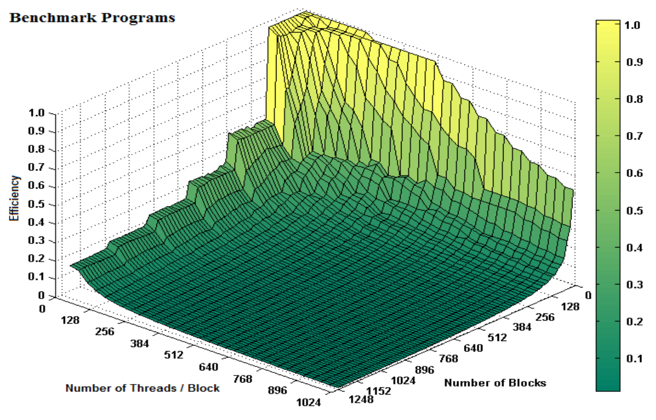
Table 7.5: Prediction Error for **first three** benchmark programs executed in Nvidia Kepler 20 GPU device.

β	**Monte Carlo π Estimation**			**Sobol Quasirandom Generator**			**Prime Number Generator**		
	EE	P_{EE}	δ_P	EE	P_{EE}	δ_P	EE	P_{EE}	δ_P
32	1.000	1.000	0.00	1.000	1.000	0.00	1.000	1.000	0.00
64	0.997	0.992	0.51	0.995	0.984	1.07	0.997	0.974	2.32
96	0.964	0.928	3.54	0.991	0.981	1.03	0.969	0.942	2.72
128	0.832	0.790	4.23	0.962	0.933	2.94	0.863	0.862	0.13
160	0.633	0.590	4.31	0.950	0.945	0.51	0.691	0.734	4.29
192	0.576	0.567	0.92	0.956	0.927	2.94	0.602	0.632	3.03
224	0.352	0.343	0.82	0.470	0.459	1.14	0.420	0.439	1.90
256	0.350	0.343	0.70	0.470	0.458	1.18	0.428	0.417	1.10
288	0.354	0.338	1.62	0.470	0.426	4.38	0.361	0.374	1.33
320	0.332	0.338	0.59	0.455	0.424	3.16	0.332	0.345	1.26
352	0.309	0.315	0.57	0.455	0.404	5.10	0.311	0.337	2.58
384	0.279	0.277	0.27	0.438	0.403	3.45	0.283	0.302	1.90
416	0.271	0.263	0.81	0.422	0.390	3.19	0.283	0.296	1.32
448	0.210	0.212	0.25	0.319	0.329	0.97	0.243	0.254	1.08
480	0.207	0.203	0.36	0.319	0.314	0.55	0.220	0.238	1.76
512	0.201	0.202	0.09	0.306	0.296	0.99	0.201	0.215	1.41

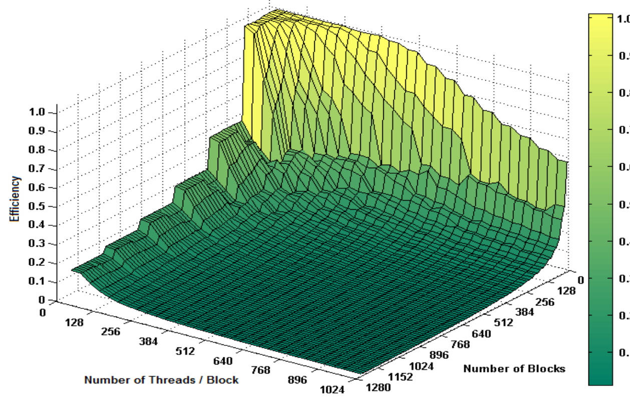
surface of Figure 7.5 (b) shows same behavior and shape as compared to the models' efficiency surface in Figure 7.4.

In this empirical work to validate the model, all six benchmark programs are executed and efficiency values are measured. Figure 7.6 (a, b, c) shows the efficiency surfaces of the Monte Carlo Estimation of π, Sobol Quasirandom Generator, and Stirling's approximation error benchmark programs. Along with the surfaces, empirical results of all benchmark programs are placed in Tables 7.5 and 7.6 for analysis. It can be observed from Figure 7.6 (a, b, c) that these surfaces depicts similar behavior when the value of β and ϑ increases. As expected, significant degradation in efficiency occurs when total spawned threads are greater than the total occupancy of each scheduling cycle. It is apparent that the theoretically derived model introduced in this chapter do estimate (predict) the actual measured efficiency surface very well.

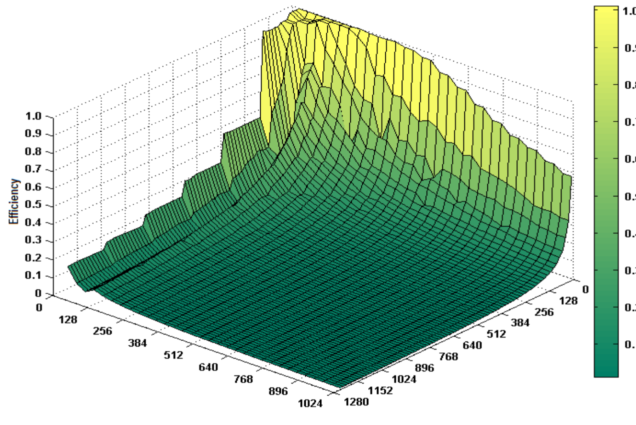
For further reporting, Tables 7.5 and 7.6 show measured execution efficiency (EE) values for all six benchmark programs along with estimated model efficiency



(a)



(b)



(c)

Figure 7.6: The efficiency surface diagram for (a) Monte Carlo Estimation of π. (b) Sobol Quasirandom Generator. (c) Stirling's Factorial Approximation Error. The number of blocks are varied from 16 to 1248 and the number of threads in a block are varied from 32 to 1024. A total of 2496 test run have been conducted for each case to measure actual efficiency of the benchmark program.

Table 7.6: Prediction Error for **next three** benchmark programs executed in Nvidia Kepler 20 GPU device.

β	**Sum of Binomial Series**			**Stirling's Factorial Appox. Error**			**CUDA FFT**		
	EE	P_{EE}	δ_P	EE	P_{EE}	δ_P	EE	P_{EE}	δ_P
32	1.000	1.000	0.00	1.000	1.000	0.00	1.000	1.000	0.00
64	0.997	0.992	0.51	0.953	0.942	1.11	0.993	0.982	1.10
96	0.922	0.901	2.19	0.726	0.746	1.98	0.996	0.976	1.99
128	0.823	0.784	4.15	0.560	0.582	2.24	0.968	0.973	0.54
160	0.657	0.639	2.32	0.521	0.518	0.34	0.863	0.882	1.95
192	0.561	0.552	1.27	0.462	0.486	2.39	0.745	0.768	2.28
224	0.353	0.361	1.36	0.391	0.425	3.40	0.370	0.389	1.91
256	0.348	0.346	0.35	0.351	0.381	3.04	0.370	0.427	5.69
288	0.349	0.318	2.61	0.302	0.327	2.50	0.426	0.454	2.76
320	0.322	0.304	1.91	0.288	0.291	0.33	0.421	0.446	2.46
352	0.306	0.293	1.44	0.269	0.261	0.77	0.412	0.428	1.62
384	0.275	0.276	0.14	0.251	0.232	1.90	0.373	0.385	1.23
416	0.264	0.253	1.28	0.232	0.216	1.59	0.373	0.381	0.85
448	0.218	0.226	1.40	0.219	0.208	1.14	0.247	0.269	2.17
480	0.212	0.201	0.56	0.193	0.174	1.90	0.271	0.272	0.15
512	0.195	0.193	0.72	0.184	0.168	1.63	0.269	0.267	0.20

(P_{EE}) values, and the prediction error ($\delta = |EE - P_{EE}| \times 100$). Values of both EE and P_{EE} are in a scale of $0.0-1.0$ and δ is in 0.0 to 100%. It can be observed from the Table 7.5 and 7.6 that the introduced efficiency prediction model achieves very low prediction error. The minimum prediction error is 0.13% for the cases in which 128 thread blocks are spwanned for the Prime Number Generator benchmark program (when $\beta \times \vartheta <$ occupancy, the prediction value is τ, thus incures no prediction error and not considered). The maximum prediction error is 5.69% in which 256 thread blocks are spawned for CUDA FFT benchmark program. Regardless of real-world benchmark programs, the prediction error incured by the model is always below 6.0% in the dynamic changing environment. The empirical results presented in both Table 7.5 and 7.6 shows the accuracy and reliability of the introduced prediction model. These empirical results justify the validity of the introduced prediction model. The proposed model and analysis can be utilized as a guideline for programmers to arrange their blocks and threads to achieve optimal execution time.

7.5 Summary

An approach to efficiently allocate a massive number of thread blocks for parallel execution in GPU devices (independent of architecture) is presented. Given a set of compute-intensive threads, and a host node with a GPU device, our goal is to find an assignment of threads on the GPU such that the total time taken to execute all the threads is minimized. The key challenge was to determine the arrangement of blocks and threads in a block prior to placement of threads into the run queue.

This chapter has introduced a prediction model to forecast the execution efficiency of GPUs for computationally intensive kernels. The model has been validated with six professionally developed benchmark programs provided in Nvidia CUDA GPU SDK. The provided surface diagrams depict clear visualization of measured efficiency based on variable number of blocks and size of blocks. The empirical studies performed on the prediction model show that the model surface follows the same shape and pattern of the real-world benchmark programs.

The empirical case study results show that the forecast provided by the prediction model is very reliable with only $0.13 - 5.69\%$ prediction error. The model uses only a few parameters like the number of blocks, size of blocks, number of streaming multi-processors, and number of resident threads. This model is significant from the perspective that it can also be utilized as a reference by programmers to arrange their blocks and threads to achieve optimal results. The conducted study is highly useful for understanding and optimizing performance on a GPU. In addition, the study could be used in the context of heterogeneous environments where multiple GPU devices with different hardware configurations are used. Specifically, the model can help dynamically to choose the device with a better performance potential.

All the obtained results justify the strength of the introduced model for predicting the execution efficiency of compute-intensive GPU kernels independent of GPU architectures. Hence, the ability of introduced model to predict the efficiency of thread execution while the arrangement is variable has been demonstrated. Thus, the usefulness of using the introduced model in real-world applications for task assignment has been motivated.

Chapter 8

Conclusion

8.1 Summary

In this dissertation, a set of prediction models for estimating CPU, memory, and GPU resource availability are introduced and a comprehensive set of evaluations are performed to verify the accuracy of the prediction models. For estimating the availability of CPU for a batch of tasks, a CPU availability prediction model is introduced and empirically verified for accuracy in both Windows systems running JVM and Linux systems with the C language. CPU availability scatter plots provide a clear visualization of measured performance based on the density of the dots in the plot. These empirically measured availability values are showed to generally fall within theoretically derived upper and lower bound formulas in both single- and multi-core architectures and consistent with the dynamic nature of system and its workload.

Because of the potential for a tighter lower-bound, observed from the empirical results of multi-core Unix systems, a new CPU availability lower-bound model is proposed which significantly reduced the bound difference. A 90% confidence interval for measured availability is shown to provide significantly tighter upper- and lower-limits than the theoretically derived models for upper- and lower-bounds as well. Based on the empirical results for validating the CPU availability model, an average CPU availability model is proposed and the empirical results show that the prediction error ranges only between 0.19% to 3.57%. It was demonstrated that shifting the relative phasing of the threads to reduce possible work phase overlap can improve the performance (i.e., CPU efficiency). It is also shown that if there are I/O bound processes in the run queue, the traditional RR model fails to accurately predict the CPU assignment as it does not have a clear idea about the unused processing time left behind by the I/O bound processes.

In addition to CPU availability, the total amount of memory required by concurrent threads is a factor in predicting the thread execution efficiency. When the total memory requirement is higher than the total available memory, the relative performance can have a significant impact on execution time. Analytical models are developed for predicting (and measuring) the execution efficiency of concurrent threads as a function of memory availability. The model is then verified with memory intensive real-world benchmark programs like merge sort, dense matrix multiplication, and linear search for measuring the accuracy. Next, the CPU availability and memory model is deployed non-trivially to derive a composite prediction model. The composite prediction model is validated empirically by an extensive set of case studies involving CPU and memory intensive real-world benchmark programs as well. The proposed composite models can be highly useful for determining the order in which tasks should be assigned to the system so that the completion time of all the tasks is minimized.

By utilizing the proposed composite prediction model, two task assignment models for dispatching tasks in heterogeneous compute nodes are proposed. The goal is to find an assignment of tasks to compute nodes such that the total time taken to execute one or a batch of tasks is minimized. The TASD model requires query to all compute nodes each time a task is to be assigned to gather nodes' status information. While such current information gives precise input to the model, it is expensive (in terms of overload) when we consider a large number of compute nodes.

To overcome this limitation, the TADT model maintains a dynamic prediction table for holding the measured CPU and memory availability estimates for each node of the distributed system. The same prediction models of TASD are used but the resource state parameter values are taken from the prediction table instead of fetching it from nodes each time before task assignment. A relative comparison among TORQUE manager (which uses Round Robin scheduling with FCFS queue), TASD, and TADT were carried out in this dissertation to demonstrate the prediction accuracy of the proposed models. An extensive set of empirical studies have been conducted in a distributed environment with heterogeneous compute nodes with real-world benchmark programs. The bar graphs and surface plots provide a clear visualization of measured execution efficiency and task completion time. It can be observed from the surface plots and bar graphs that the introduced TASD and TADT models clearly outperforms the TORQUE scheduler. The new CPU usage prediction model used in the TADT model provides accurate predicted CPU availability for incoming tasks to make intelligent task dispatch decision to distributed nodes.

An approach to efficiently allocate a massive number of thread blocks for parallel execution in GPU devices (independent of architecture) is introduced. Given a set of compute-intensive threads, and a compute node hosting a GPU device, our goal is to find an assignment of threads on the GPU such that the total time taken to execute all the threads is minimized. The key challenge was to determine the arrangement of blocks and threads in a block prior to placement of threads into the run queue.

Besides empirical work in distributed environment, extensive work has been performed in parallel environment to introduced prediction models to forecast the execution efficiency of GPUs for computationally intensive kernels. The model has been validated with six professionally developed benchmark programs provided in Nvidia CUDA GPU SDK. The provided surface diagrams depicts clear visualization of measured efficiency based on variable number of blocks and size of blocks. The empirical studies performed on the prediction model show that the model surface follows the same shape and pattern of the real-world benchmark programs. The empirical case study results show that the forecast provided by the prediction model is very accurate with only $0.13 - 5.69\%$ prediction error. This model is significant from the perspective that it can also be utilized as a reference by programmers to arrange their blocks and threads to achieve optimal results. The conducted study is highly useful for understanding and optimizing performance on a GPU. In addition, the study could be used in the context of heterogeneous environments where multiple GPU devices with different hardware configurations are used. The models could help dynamically to choose the device with a better performance potential.

All the obtained results of empirical work justify the strength of the introduced model for predicting the efficiency of multi-core and many-core systems while executing compute- and memory-intensive programs. Hence, the ability of the introduced models to predict the efficiency of thread execution while resources and arrangement are variable has been demonstrated. The usefulness of using the introduced models in real-world applications for predicting resource availability and task assignment has been demonstrated.

8.2 Future Work

The work presented in this dissertation can be further extended as follows.

- All the CPU and memory efficiency prediction related work in this dissertation is validated with real single- and multi-core systems. Recently, the use of *hypervisor*, which enables the running of multiple operating systems in a single machine, has increased. So, there is future to analyze the CPU and memory management in virtual machines and possible validation (with some additional parameter/fine tuning) of our introduced CPU and memory model in virtual machine environments.

- Solid State Drive (SSD) technology is getting popular, which provides faster read and write operation compared with the traditional hard-disks. In our memory performance model, we haven't used the swap time possible with SSD. Moreover, the SSD provides almost no backing store seek time or latency which should improve the thread execution efficiency.

- The introduced CPU usage prediction model can be deployed in cloud environment. The model can dynamically measure available resources when a new task arrives or an old task finishes execution. Task preemption and distribution in cloud environment using the introduced CPU usage prediction model can be considered as well.

- The introduced TASE model provides accurate results but we did not check the accuracy for longer period of time or a run queue consisting of more than 30 tasks. The TASE model may require periodic adjustment (depending on time or number of dispatched tasks) which needs to be analyzed and determined.

- The introduced GPU performance model can be utilized to implement an API that can provide automatic suggestions to programmers to rearrange blocks and threads in a grid to utilize the full potential of the GPU. That API can be integrated with the CUDA or Open MP compiler for automatic arrangement optimization to improve the GPU utilization.

- The introduce GPU performance model is only suitable for compute-intensive kernels. In reality, we can have kernels which require significant memory access as well. The model needs to be extended to incorporate the effect of the global memory and other associated overhead.

- The introduced CPU and memory model can be integrated with the GPU performance model for use in large heterogeneous systems. Because of the trend towards GPU accelerated applications to boost performance of GPGPU applications, this topic is very interesting and challenging.

Reference List

[1] Martha Beltrań, Antonio Guzmań and Jose Luis Bosque, “A new CPU Availability Prediction Model for Time-Shared Systems”, *IEEE Transactions on Computers, Vol 57, No. 7, pp. 865-875,* July 2008.

[2] Khondker S. Hasan, John K. Antonio, and Sridhar Radhakrishnan, “A New Composite CPU/Memory Model for Predicting Efficiency of Multi-core Processing”, *The 20th IEEE International Symposium on High Performance Computer Architecture (HPCA-2014)* workshop , Sponsored by: IEEE Computer Society, Orlando, FL, February 15-19, 2014.

[3] Jaime Montoya, “Linux Operating System”, URL: http://www.monografias.com/trabajos43/linux-operating-system/linux-operating-system.shtml

[4] Bill Venners, “Inside the Java 2 Virtual Machine”, Thread Synchronization, URL: http://www.artima.com/insidejvm/ed2/index.html

[5] Jingtong Hu, Yi He, Qingfeng Zhuge, Edwin H.-M. Sha, Chun Jason Xue, and Yingchao Zhao, “Minimizing accumulative memory load cost on multi-core DSPs with multi-level memory Original Research Article”, *Journal of Systems Architecture,* Elsevier, vol. 59, issue 7, pp. 389–99, 2013.

[6] Tyler Dwyer, Alexandra Fedorova, Sergey Blagodurov, Mark Roth, Fabien Gaud, Jian Pei, “A Practical Method for Estimating Performance Degradation on Multicore Processors, and its Application to HPC Workloads”, *SC ’12 Proc.*

of the Int. Conference on High Performance Computing, Networking, Storage and Analysis, Article No. 83, ISBN: 978-1-4673-0804-5, IEEE Computer Society Press Los Alamitos, CA, 2012.

[7] Khondker Shajadul Hasan, Nicolas G. Grounds, John K. Antonio, “Predicting CPU Availability of a Multi-core Processor Executing Concurrent Java Threads”, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 11),* sponsor: World Academy of Science and Computer Science Research, Education, and Applications (CSREA), Las Vegas, NV, July 2011.

[8] M. Beltrán and J. L. Bosque, “Estimating a Workstation CPU Assignment with the DYPAP Monitor”, *Proc. Third IEEE International Symposium on Parallel and Distributed Computing*, pp. 64-70, ISBN: 0-7695-2210-6, August 2004.

[9] Godofredo R. Garay, Julio Ortega, Antonio F. Díaz, Luis Corrales, and Vicente Alarcón-Aquino, “System performance evaluation by combining RTC and VHDL simulation: A case study on NICs Original Research Article”, *Journal of Systems Architecture,* Elsevier, vol. 59, issue 10, part D, pp. 1277-98, 2013.

[10] Thomas Rauber, Gudula Rünger, “Anticipated Distributed Task Scheduling for Grid Environments”, *20th International Parallel and Distributed Processing Symposium, IPDPS 2006,* ISBN: 1-4244-0054-6,Rhodes Island, USA, April 2006.

[11] Y. Zhang, W. Sun, and Y. Inoguchi, “Predicting running time of grid tasks on CPU load predictions”, *Proc. 7th IEEE/ACM International Conference on Grid Computing, pp. 286-292,* Sept. 2006.

[12] Alexandra Fedorova, David Vengerov, Daniel Doucette, *"Operating System Scheduling On Heterogeneous Core Systems"',* Sun Microsystems Technical Report, http://www.techrepublic.com/whitepapers/operating-system-scheduling-on-heterogeneous-core-systems/314436, July 2007.

[13] Silberschatz Avi, Galvin B. Peter, Gagne Greg, *Operating System Concepts,* Eighth Edition, John Wiley and Sons, Jan 26, 2010.

[14] Andrew S. Tanenbaum, *Modern Operating Systems*, Third Edition, ISBN13: 978-0136006633, Prentice Hall Inc., 2008.

[15] Hira K. Shrestha, Nicolas Grounds, Jason Madden, Matthew Martin, John K. Antonio, Jay Sachs, Josh Zuech, and Carlos Sanchez, Scheduling Workflows on a Cluster of Memory Managed Multicore Machines, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 09),* Las Vegas, NV, July 2009.

[16] M. Beltrán and Antonio Guzmán, "How to Balance the Load on Heterogeneous Clusters", *International Journal of High Performance Computing Applications,* Volume 23, No. 1, pp. 99118, Spring 2009.

[17] Khondker S. Hasan, Sridhar Radhakrishnan, and John K. Antonio, "Composite Prediction Model and Task Distribution on a Cloud of Multi-core Processors", *The IEEE Conference on High Performance Computing (HiPC-2013) workshop on Cloud Computing Applications (IWCA-13),* Bangalore, India, 18-21 Dec. 2013.

[18] Andrea Arcuri, A Theoretical and Empirical Analysis of the Role of Test Sequence Length in Software Testing for Structural Coverage, *IEEE Transactions on Software Engineering,* Vol. 38, pp. 497-519, issn 0098-5589, 2012.

[19] "Linux scheduling summary", http://www.programering.com/a/MjN5ITMwATU.html, China, March 10, 2014.

[20] Completely Fair Scheduler (CFS), http://en.wikipedia.org/wiki/Completely_Fair_Scheduler, 5th May, 2014.

[21] Václav Chlumský, Dalibor Klusacek, and Miroslav Ruda, "The Extension of TORQUE Scheduler Allowing the Use of Planning and Optimization in Grids", *Computer Science Journal, Vol 13, No 2,* DOI: 10.7494/csci.2012.13.2.5, 2012.

[22] Wan Yeon Lee, Energy-Efficient Scheduling of Periodic Real-Time Tasks on Lightly Loaded Multicore Processors, *IEEE Transactions on Parallel and Distributed Systems,* vol. 23 no. 3, pp. 530-537, issn 1045-9219, 2012.

[23] C.-H. Philip Yuen and S.H. Gary Chan, Scalable Real-Time Monitoring for Distributed Applications, *IEEE Transactions on Parallel and Distributed Systems,* vol. 23 no. 12, pp. 2330-2337, issn 1045-9219, 2012.

[24] Vinay G. Vaidya, Sudhakar Sah and Priti Ranadive, Optimal Task Scheduler for Multi-core Processor, *2nd International Conference on Software Technology and Engineering(ICSTE),* Center for Research in Engineering Sciences ad Technology, KPIT Cummins Infosystems Ltd., Pune, Maharashtra, India, 2010.

[25] Nicolas G. Grounds, John K. Antonio, and Jeff Muehring, "Cost-Minimizing Scheduling of Workflows on a Cloud of Memory Managed Multicore Machines", *CloudCom 2009,* LNCS 5931, pp. 435-450, 2009.

[26] Rafael H. Saavedra, David E. Culler, Thorsten Eicken, "Analysis of Multithreaded Architecture for Parallel Computing", *2nd Annual ACM Symposium on Parallel Algorithms and Architectures,* ISBN: 0-89791-370-1, 1990.

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms,* Ed. 4, Prentice-Hall, 2009.

[28] Kunz T., "The influence of different workload descriptions on a heuristic load balancing scheme", *IEEE Transactions on Software Engineering*, Vol. 17, Issue 7: 725-730, 1991.

[29] Niklas Carlsson, Carey Williamson, Andreas Hirt, Michael Jacobson Jr. "Performance modeling of anonymity protocols", *Journal of Performance Evaluation,* Elsevier, Vol. 69, no. 12 , pp. 643-661, Dec. 2012.

[30] P.A. Dinda, "Online Prediction of the Running Time of Tasks", *Proc. 10th IEEE International Symposium High Performance Distributed Computing,* pp. 336-7, 2001.

[31] R. Wolski, N. Spring, and J. Hayes, "Predicting the CPU Availability of Time-Shared Unix Systems on the Computational Grid," *Proc. 8th International Symposium on High Performance Distributed Computing, pp. 105-112,* ISBN: 0-7803-5681-0, August 2002.

[32] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, Keith I. Farkas, "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance", *31st Annual International Symposium on Computer Architecture (ISCA),* June 2004.

[33] M. Iverson, F. Ozguner, and G. Follen, Run-Time Statistical Estimation of Task Execution Times for Heterogeneous Distributed Computing, *"Proc. High Performance Distributed Computing Conference",* pp. 263-270, 1996.

[34] TARINGA, *"Types of processes in Linux"*, Linux and GNU http://www.taringa.net/posts/linux/14121428/Tipos-de-procesos-en-Linux.html, 2012.

[35] Avinesh Kumar, *"Multiprocessing with the Completely Fair Scheduler for Linux",* IBM Technical Report, http://www.ibm.com/developerworks/linux/library/l-cfs/author1 Jan 2008.

[36] M. Tim Jones, *"Inside the Linux scheduler 2.6",* The Journal Of A Linux Sysadmin, http://www.ducea.com/2006/07/08/inside-the-linux-scheduler/, July 2006.

[37] Ana M. Morenoa, Maria-Isabel Sanchez-Segurab, Fuensanta Medina-Dominguezb, Laura Carvajala, "Balancing software engineering education and industrial needs", *Journal of Systems and Software,* Vol. 85, Issue 7, pp. 1607-1620, July 2012.

[38] J. Aas, *"Understanding the Linux 2.6.8.1 CPU Scheduler",* Silicon Graphics Intl, http://josh.trancesoftware.com/linux/linux-cpu_scheduler.pdf, 2005.

[39] M. Tim Jones, *"Inside the Linux scheduler",* The latest version of this all-important kernel component improves scalability, IBM Technical Report, http://www.ibm.com/developerworks/linux/library/l-scheduler/, June 2006.

[40] Randal E. Bryant, David R. O'Hallaron, *Computer Systems, A Programmer's Perspective*, Second Edition, Prentice Hall Inc., 2011.

[41] P. Mehra and B.W. Wah, "Automated Learning of Workload Measures for Load Balancing on a Distributed System," Proc. Int'l Conf. Parallel Processing, Volume 3: Algorithms and Applications, pp. 263-270, 1993.

[42] Khondker Shajadul Hasan "A Distributed Chess Playing Software System Model Using Dynamic CPU Availability Prediction", sponsor: World Academy of Science and Computer Science Research, Education, and Applications (CSREA), *International Conference on Software Engineering Research and Practice (SERP-11),* Las Vegas, Nevada, July 2011.

[43] Paessler Knowledge Base, *"Monitoring Running Processes in Linux"*, http://www.paessler.com/knowledge-base/en/topic/29403-monitoring-processes-in-linux, Dec 2011.

[44] Brian K. Tanaka, *"Monitoring Virtual Memory with vmstat"*, Linux Journal, http://www.linuxjournal.com/article/8178, Oct 31, 2005.

[45] Roberto Espinoza, *"Process Scheduling in Linux"*, CPC Computer Consultants, http://www.cpccci.com/blog/2009/01/28/process-scheduling-in-linux/, Jan 2009.

[46] M. Beltran and J.L. Bosque, "Predicting the Response Time of a New Task on a Beowulf Cluster", *Lecture Notes in Computer Science*, R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Wasniewski, eds., vol. 3019, pp. 145-152, Springer, 2003.

[47] Rares Aioanei, *"Linux Process Scheduling Made Easier"*, News in LinuxCareer.com, http://how-to.linuxcareer.com/linux-process-scheduling-made-easier-lance, 17 October 2011.

[48] Kevin O Brien, *"Western Digital VelociRaptor 1TB Review"*, Review report in storagereview.com, http://www.storagereview.com/western-digital-velociraptor-1tb-review, April 16, 2012.

[49] A. Snavely and D. Tullsen, "Symbiotic job scheduling for a simultaneous multi-threading architecture", *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, Pages 234-44, New York, Nov. 2000.

[50] M. A. Bender, and M. O. Rabin, "Scheduling Cilk multithreaded parallel programs on processors of different speeds", *12th ACM Sym. on Parallel Algorithms and Architectures*, pp. 13-21, July 2000.

[51] Shannon Cepeda, *"Intel Hyper-Threading Technology: Your Questions Answered"*, http://software.intel.com/en-us/articles/intel-hyper-threading-technology-your-questions-answered 27 January, 2012.

[52] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. W. Hwu, "An adaptive performance modeling tool for GPU architectures", *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010),* pp. 10517114, Jan. 2010.

[53] Sim, Jaewoong and Dasgupta, Aniruddha and Kim, Hyesoon and Vuduc, Richard, "A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications", *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12),* pp. 11-22, New Orleans, Louisiana, 2012.

[54] Zhou S., "A trace-driven simulation study of dynamic load balancing", *IEEE Transactions on Software Engineering*, pages 1327-1341, 1988.

[55] Garrick Staples, "TORQUE resource manager", *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing* ,ISBN 0-7695-2700-0, Tampa, FL, November 11 - 17, 2006.

[56] Adaptive Computing, "TORQUE Resource Manager" http://www.adaptivecomputing.com/products/open-source/torque/

[57] John K. Holmen and David L. Foster, "Accelerating Single Iteration Performance of CUDA-Based 3D ReactionDiffusion Simulations", *International Journal of Parallel Programming,* Springer, Volume 42, Issue 2, pp. 343-363, DOI 10.1007/s10766-013-0251-z, April 2014.

[58] Chatterjee, Amlan and Radhakrishnan, Sridhar and Antonio, John K, "Data Structures and Algorithms for Counting Problems on Graphs using GPU", *International Journal of Networking and Computing,* vol. 3, num. 2, pp. 264-288, 2013.

[59] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu, “Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA”, *The 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP-08),* Salt Lake City, Utah, February 201723, 2008.

[60] Rob Williams, *”A Look at NVIDIAs Kepler-based Tesla K-Series GPU Accelerators”*, http://techgage.com/article/a-look-at-nvidias-kepler-based-tesla-k-series-gpu-accelerators/, November 12, 2012.

[61] Jeremiah van Oosten, *“Introduction to CUDA 5.0 — 3D Game Engine Programming”,* http://3dgep.com/?p=4151#Kepler_Architecture, Oct 26, 2012.

[62] Sara Baghsorkhi, Wen-mei Hwu, “Analytical Performance Prediction for Evaluation and Tuning of GPGPU Applications”, *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPHAM 2009),* Seattle, Washington. March 22-25, 2009.

[63] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S. Z. Ueng, S. S. Baghsorkhi, and W. W. Hwu, “Program optimization carving for GPU computing”, *Journal of Parallel and Distributed Computing,* vol. 68, no. 10, pp. 1389171401, Oct. 2008.

[64] Weiguo Liu and Bertil Schmidt, “Performance Predictions for General-Purpose Computation on GPUs”, *IEEE International Conference on Parallel Processing (ICPP 2007),* Xian, China, Sept 2007.

[65] Yao Zhang and Owens, J.D., “A quantitative performance analysis model for GPU architectures”, *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on,* pp. 382-393, 2011.

[66] Chatterjee, Amlan and Radhakrishnan, Sridhar and Antonio, John K "On Analyzing Large Graphs Using GPUs", *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pp. 751-760, 2013.

[67] Al-Mouhamed, M. and ul Hassan Khan, A., "Exploration of automatic optimization for CUDA programming", *Parallel Distributed and Grid Computing (PDGC), 2012 2nd IEEE International Conference on,* pp. 55-60, 2012.

[68] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of PTX kernels", *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC 2009),* pp. 31712, Oct. 2009.

[69] Changmin Lee, Won Woo Ro, and Jean-Luc Gaudiot, "Boosting CUDA Applications with CPUGPU Hybrid Computing", *International Journal of Parallel Programming,* Springer, Volume 42, Issue 2, pp. 384-404, DOI 10.1007/s10766-013-0252-y, April 2014.

[70] U. Chandra and M. Harmon, "Predictability of Program Execution Times on Superscalar Pipelined Architectures", *Proc. Third Workshop Parallel and Distributed Real Time Systems*, pp. 104-113, 1995.

[71] C.Y. Park and A.C. Shaw, "A Source-Level Tool for Predicting Deterministic Execution Times of Program", Technical Report 89-09-12, Univ. of Washington, 1989.

[72] S. Hong and H. Kim, "An Analytical model for a GPU architecture with memory-level and thread-level parallelism awareness", *Proceedings of the 36th International Symposium on Computer Architecture (ISCA 2009),* pp. 15217163, Jun. 2009.

[73] Nvidia CUDA GPU SDK, Sample CUDA Toolkits, http://docs.nvidia.com/cuda/cuda-samples/

[74] Chatterjee, A. and Radhakrishnan, S. and Antonio, John K., "Counting Problems on Graphs: GPU Storage and Parallel Computing Techniques", *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International,* pp. 804-812, Shanghai, China, 2012.

[75] Mark Harris, *"How to Implement Performance Metrics in CUDA C/C++"*, http://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/, November 7, 2012.

[76] H. Dail, F. Berman, and H. Casanova, A decoupled scheduling approach for Grid application development environments, *Journal of Parallel and Distributed Computing,* 63:505 524, 2003.

[77] Z. Liang; Y. Sun and C. Wang, "ClusterProbe: an open, flexible and scalable cluster monitoring tool", *In Proceedings of the First International workshop on Cluster Computing*, pages 261-268, 1999.

[78] *"TESLA GPU Accelerators for workstations"*, http://www.nvidia.com/object/nvidia-kepler.html, 2012.

[79] *"Graphics processing unit (GPU)"*, http://en.wikipedia.org/wiki/GPU

[80] *"Kepler microarchitecture"*, http://en.wikipedia.org/wiki/Kepler_(microarchitecture)

[81] *"CUDA: Compute Unified Device Architecture"*, http://en.wikipedia.org/wiki/CUDA

[82] Eunjung Park, John Cavazos, Louis-Nol Pouchet, Cédric Bastoul, Albert Cohen, and P. Sadayappan, "Predictive Modeling in a Polyhedral Optimization Space", *International Journal of Parallel Programming,* Springer, Volume 41, Issue 5, pp. 704750, October 2013.

[83] *"Kepler Compute Architecture Technology in a Brief"*, http://www.nvidia.com/content/PDF/kepler/NV_DS_Tesla_KCompute _Arch_May_2012_LR.pdf, 2012.

[84] *"Kepler - The World's Fastest, Most Efficient HPC Architecture"*, http://www.nvidia.com/object/nvidia-kepler.html, 2012.

[85] NVIDIA Corporation, *"CUDA C Programming Guide. (PG-02829-001_v5.0)"*, Available from: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, Accessed: October 2012.

[86] NVIDIA Corporation, *"NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110. (V1.0)"*, Available from: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, Accessed: October 2012.

[87] CUDA Toolkit Documentation, Parallel Thread Execution ISA Version 4.0, http://docs.nvidia.com/cuda/parallel-thread-execution/#axzz369L7GrMv

[88] CUDA Warps and Occupancy, *"GPU Computing Webinar"*, Available from: http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf, July 2011.

[89] Jen-Chieh Yeh; Chi-Hung Lin; Chun-Nan Liu, "Multi-core system performance prediction and analysis at the ESL", *Int. J. of Computational Science and Engineering,* DOI: 10.1504/IJCSE.2014.058700, Vol. 9 No. 1/2, pp. 86–94, 2014.

[90] Siu Kwan Lam, *"CUDA Performance: Maximizing Instruction-Level Parallelism",* Available from: http://continuum.io/blog/cudapy_ilp_opt, September, 2013.

[91] *"How CUDA Blocks/Warps/Threads map onto CUDA Cores?"*, Answered by: Greg Smith, Available from: http://stackoverflow.com/questions/10460742/how-cuda-blocks-warps-threads-map-onto-cuda-cores, May 2012.

Appendix A

Definitions

- **CPU availability:** the percentage of CPU time that will be allocated for a new task.
- **CPU-bound process:** the process that is always ready to run in the processor. Therefore, it consumes its quantums, or slices of CPU time, completely and it can always take advantage of all the available CPU time.
- **CPU requirement:** the fraction of time a task requires the CPU resource.
- **CPU usage:** the fraction of time a thread spends utilizing CPU resources on an unloaded system.
- **Execution efficiency:** the amount of CPU time that a task requires to complete its execution when adequate free memory frames are available in main memory divided by the total time it takes when only part of memory frames is available in the main memory.
- **I/O-bound process:** the process that cannot take advantage of all the CPU time that corresponds it with the Round Robin scheduling, either because it is not always ready to run or because it does not finish all its quantums.

- **Memory requirement:** the maximum amount of main memory a task will consume at any point during its execution.

- **Memory availability percentage (MAP):** the percentage of available free primary memory with respect to the total memory requirements by all threads in a batch to be spawned concurrently.

- **Response time:** the total time between start and end time of a task(thus, it includes processor and system times).

- **Single Instruction Multiple Thread (SIMT):** the class of parallel computing that describes computers with multiple processing elements that perform the same operation on multiple threads simultaneously. Thus, such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment.

- **Single program multiple data (SPMD):** the technique employed to achieve parallelism. Tasks are split up and run simultaneously on multiple processors with different input in order to obtain results faster. SPMD is the most common style of parallel programming.

- **Thread Divergence:** if a kernel contains divergence (multiple execution paths), different set of threads within the same warp will follow different control paths. In that situation, executing a warp requires several passes; one for each control path leading to variable execution time.

- **Unloaded-CPU usage:** the CPU time consumed by a process say P_i during its execution divided by its response time, both with the processor completely unloaded.

- **Warp occupancy:** is defined as the number of active warps divided by the maximum active warps in a streaming multi-core processor of a GPU.

Appendix B

Acronyms

DYPAP Dynamic Process Assignment Prediction Model

HT Hyper-threading of a CPU core

PS Phase shifting

CC CPU Cores

CUF CPU usage factor

DHCP Dynamic Host Configuration Protocol

DNS Domain Name System

FTP File Transfer Protocol

FCFS First Come First Served

GPU Graphical Processing Unit

HIP Host Identification Protocol

ICMP Internet Control Message Protocol

IP Internet Protocol

JVM Java Virtual Machine

LLM Local Location Manager

MAM Memory Availability Model

MAP Memory Availability Percentage

NRT Non-Real Time

RR Round Robin Scheduling Model

RT Real Time

RTP Response Time Prediction

RTT Round Trip Time

SIMD Single Instruction Multiple Data

SMP Symmetric multi-processing

SPAP Static Process Assignment Prediction Model

SPMD Single program multiple data

SSPAP Simplified Static Process Assignment Prediction Model

SM Streaming Multi-core Processor

TADT Task Assignment using Dynamic Table

TASD Task Assignment using Status Data of Compute Node

TCP Transmission Control Protocol

TLB Translation Look-aside Buffer

TORQUE The Terascale Open-source Resource and QUEue Manager

UDP User Datagram Protocol