A THEORETICAL FRAMEWORK FOR DESIGN SPACE EXPLORATION OF MANYCORE PROCESSORS

by

HUN JUNG

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2011

ACKNOWLEDGEMENTS

First of all, I want to express my deepest appreciation to Dr. Hao Che. Without his careful guidance, encouraging advises, I would have been in trouble keeping my research going to its completion. His strong theoretical and practical knowledge of the network and system architectures have led me to obtain better solutions in various problems I have faced. I would also like to thank all the committee members: Dr. Roger Walker, Dr. Mohan Kumar, Dr. Ramez Elmasri for valuable advises on my research.

I also want to tell all my friends that I have enjoyed their personalities and supports. In particular, I am thankful for the enjoyable friendship and valuable discussions I have had with David Levine and Miao Ju throughout the years.

Finally, I'd like to express my deep appreciation for my parents' endless support and care. Last but not the least, I would like to thank my wife and my two princesses for their constant patience, love and support.

October 27, 2011

ABSTRACT

A THEORETICAL FRAMEWORK FOR DESIGN SPACE EXPLORATION OF MANYCORE PROCESSORS

Hun Jung, PhD

The University of Texas at Arlington, 2011

Supervising Professor: Hao Che

As design space and workload space in multicore era are continuously expanding, it is a challenge to identify optimal design points quickly during the early stage of multicore processor design or programming phase. To meet this challenge, a thread-level modeling methodology is developed in this dissertation. The idea is to model multicore processors at the thread-level, overlooking instruction-level and microarchitectural details. Since the thread-level modeling is much coarser than the instruction-level modeling, the analysis at this level turns out to be significantly faster than that at the instruction level. This feature makes the methodology particularly amenable to fast performance evaluation for manycore systems in a large design space.

Based on this methodology, we developed a thread-level simulation tool for quick evaluation of any given design point and also a theoretical framework that can capture the general performance properties for a class of multicore processors of interest over a large design space and workload space, free of scalability issues. In the theoretical framework, queuing network models that model multicore processors at the thread level are developed and

scalability issues in the queuing networks are solved based on an iterative algorithm over a large design space and workload space. This framework scales to virtually unlimited numbers of cores and threads.

For the simulation tool, case studies based on a large number of code samples available in IXP1200/2400 workbenches show that the maximum throughput estimated using our tool are consistently within 6% of cycle-accurate simulation results. Moreover, each simulation run takes only a few seconds to finish on a Pentium 4 computer, which strongly demonstrates the power of this tool for fast communication processor (CP) performance testing. For the theoretical frame work, the testing results demonstrate that the throughput performance for manycore processors with 1000 cores can be evaluated within a few seconds on an Intel Pentium 4 computer and the results are within 5% of the simulation data obtained based on the thread-level simulator tool.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

CHAPTER 1

INTRODUCTION

As multicore processors (MPs) become the mainstream processor technology, challenges arise as to how to design and program MPs to achieve desired performance for applications of diverse nature. This type of processors is generally built based on either specially designed processor cores, as in the case of Communication Processors (CPs) [1], or general purpose processor cores [2]. There are two scalability barriers that the existing MP analysis approaches (e.g., simulation and benchmark testing) find difficult to overcome. The first barrier is the difficulty for the existing approaches to effectively analyze MP performance as the numbers of cores and threads of execution become large. The second barrier is the difficulty for the existing approaches to perform comprehensive comparative studies of different architectures as MPs proliferate. In addition to these barriers, how to analyze the performance of various possible design/programming choices during the initial MP design/programming phase is particularly challenging, when the actual instruction-level program is not available.

To overcome the above scalability barriers, approaches that work at much coarser granularities (e.g., overlooking microarchitectural details) than the existing approaches should be sought to keep up with the ever growing design space. Such an approach should be able to characterize the general performance properties for a wide variety of MP architectures and a large workload space at coarse granularity. Moreover, such an approach should not require the availability of the instruction-level programs as input for performance analysis. The aim is to narrow down the design space of interest at coarse granularity, in which the existing approaches can work efficiently to further pin down the optimal points at finer granularities. To

this end, we believe that an overarching *theoretical* approach, encompassing both existing and future design and workload spaces, must be sought. In this dissertation, we develop a simulation tool and a theoretical framework of such kind.

A common unique feature is employed in both our simulation tool and theoretical framework to overcome the scalability barriers. First, *they works at the thread level, overlooking instruction-level and microarchitectural details, except those having significant impact on thread level performance.* A simulation tool developed at this granularity [3] was found to be capable of predicting the system performance pretty accurately, i.e., within 6% of the cycle-accurate simulation results. Also, the theoretical framework testing [52] demonstrates that the throughput performance for many-core processors with 1000 cores can be evaluated within a few seconds on an Intel Pentium 4 computer and the results are within 5% of the simulation data obtained based on the simulation tool. The performance data within 5-6 % of actual performance in this programming phase should be considered reasonably accurate. This is because such performance data is obtained using a piece of pseudo code with normally inaccurate instruction count, not the executable program itself, as input. The program developed in later phases can generally be fine tuned to compensate for such loss of accuracy. Moreover, as we shall see in Chapter 3 and 4, this granularity is particularly amenable to large design space exploration in both simulation and theoretical analysis.

The unique feature in our theoretical framework is that the approach taken for the design space exploration is unconventional. Instead of exploring the design space based on sampled points in the space, the framework directly study the general performance properties of *system classes* over the *entire* design space. Here a system class characterizes a class of multicore architectures, a workload space, and a set of performance measures. *Understanding the general performance properties of a system class leads to the understanding of the properties of all individual points in the design space* (*i.e., specific multicore architectures, specific workloads, and the associated performance data*). This approach is quite similar to

Function Analysis in mathematics that analyzes general properties of functions over the entire vector space, as illustrated in Figure 1.1. At the core of this approach is to derive the generation function $G(x)$ for a system class of interest defined in a large design space, through which all the performance measures can be further derived. In our framework, the design space and system classes are expressed mathematically using the language of queuing network models.

In this dissertation, we make the following major contributions. First, we develop a novel design methodology which is a thread-level modeling technique for multithreaded MPs. Second, based on this modeling technique, we build a simulation tool which is fast and generic. Third, we develop the theoretical framework that allows $G(x)$ to be derived for classes of multicore processors with virtually unlimited numbers of cores, overcoming the above mentioned scalability barriers.

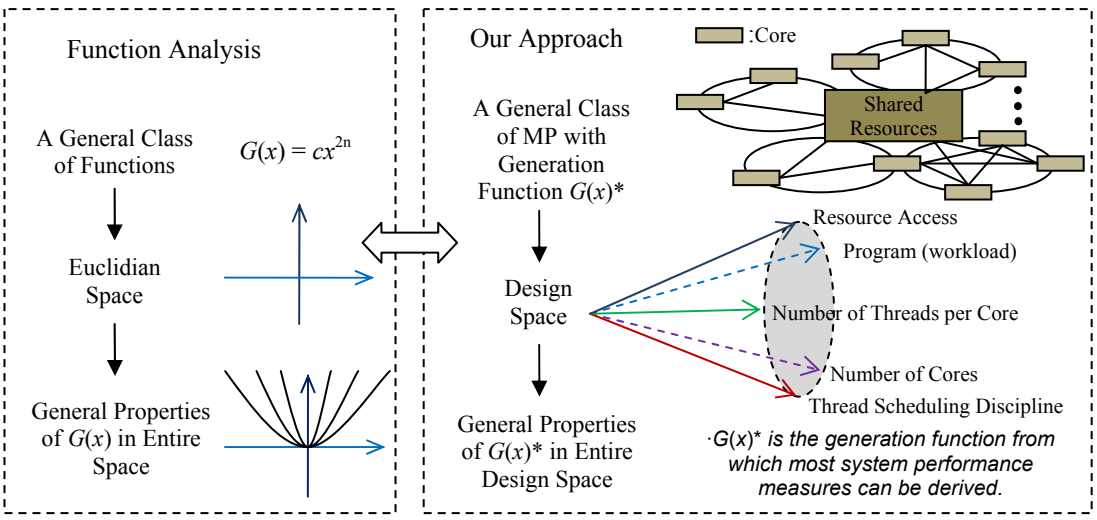

Figure 1.1 Comparison of Our Approach with Function Analysis

The rest of the dissertation is organized as follows. Chapter 2 describes the proposed methodology which is applied to both the simulation tool and the theoretical framework. Chapter 3 introduces the simulation tool. Chapter 4 shows how the theoretical framework is defined and developed. Finally, Chapter 5 concludes the dissertation.

CHAPTER 2

METHODOLOGY OF MODELING MULTICORE PROCESSORS

In this chapter, we first describe the thread-level modeling concepts and how the thread-level modeling can be mapped to queuing network models. Then we introduce a large design space that can be represented in terms of queuing network models, whose generation functions have closed-form solutions.

2.1 Thread-Level Modeling

At the core of the simulation tool and theoretical framework is the modeling of the workload, defined as a mapping of program tasks to threads in different cores and system components, known as *code paths*. As shown in Figure 2.1, a code path handled by a given thread in a given core is a sequence of segments (measured in the unit of CPU cycles) representing the durations the thread is serviced by the CPU and other resources (not including queuing delays or other idle times) throughout the execution of the entire program or program task.

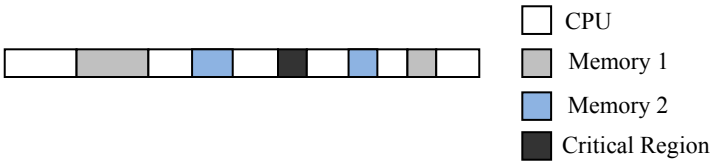

Figure 2.1 An Example of Code Path

The code path is defined at the thread level, in the sense that it only captures the events that have major impact on the thread-level performance. In other words, the instruction-level and microarchitectural details are overlooked, unless they trigger events that may have a significant effect at the thread level, such as an instruction for memory access that causes the thread to stall or instructions corresponding to a critical region that causes serialization effect at the thread level. A code path defined at this level can be easily derived from a pseudo code, rather than an instruction-level program, which may not be available during the processor design or initial programming phase.

Correspondingly, all the components including CPU, cache/memory, and interconnection network are modeled at a highly abstract level, overlooking microarchitectural details, just enough to capture the thread-level activities. For example, for a CPU running a coarse-grained thread scheduling discipline and a FIFO memory, they are modeled simply as queuing servers running a coarse-grained thread scheduling algorithm and FIFO discipline, respectively. As an example, we consider a single coarse-grained core with a FIFO memory. The core runs two active threads loaded with the same code path (on the left in Figure 2.2). The execution sequence is shown on the right in Figure 2.2. The black segments are thread idle times for the CPU.

Now consider a closed queuing network composed of two FIFO queuing servers, modeling the coarse-grained CPU and the FIFO memory, as shown in Figure 2.3 (a). Assume that there are two jobs circulating in this network, modeling the two active threads.

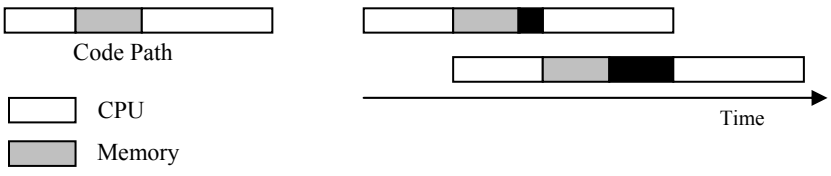

Figure 2.2 Execution Sequence for Coarse-Grained Core

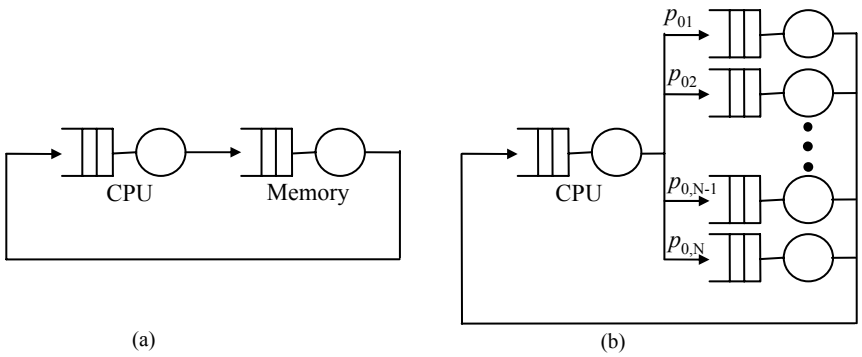

Figure 2.3 Queuing Network Models

As one can see, without considering the queuing times or thread waiting times, a thread making a round-trip, CPU-to-Memory-to-CPU, plus CPU-to-Memory, generates three segments corresponding to the code path in Figure 2.2. If the service times at the CPU and the memory exactly match the corresponding segment lengths of the code path, this queuing network model exactly emulates the execution sequence for that thread. Now, with two threads, it is not difficult to convince ourselves that due to the queuing effect, the two threads making such a trip will generate the same pattern of the execution sequence as the one on the right in Figure 2.2. Again, if the service times exactly match the segment lengths, the thread circulation exactly recovers the execution sequence in Figure 2.2.

So far we have been trying to emulate the actual execution process for the threads, which is no different from simulating the actual process at the thread level. Now we need to realize that the queuing models are in essence stochastic models, which are meant to capture long-run stochastic/statistic effects of a real system (open queuing network models may need to be used if the workload may be on and off, which however, can always be transformed into closed queuing network models [4]). In other words, the service time for a queuing server is in general a random number, following a given distribution, denoted as μ_i, for queuing server i. As a result, it is the distribution of the segment lengths, not the individual segment lengths, that

needs to be used to characterize the service time. Moreover, for a code path that characterizes a workload for a processor with multiple parallel resources, such as the one in Figure 2.1, the corresponding closed queuing network, as depicted in Figure 2.3 (b), also involves a routing probability p_{0i} for a thread to go to the i-th resource upon exiting the CPU server. This parameter should also be evaluated statistically by counting the frequency of such occurrences in the long-run code paths handled by these threads.

From the above examples, we conclude that at the thread level, any types of MPs with N components and any long-run workloads can be generally modeled as a closed queuing network with N queuing servers of various service types in terms of queue scheduling disciplines and a workload space ($\{\mu_i\}$, $\{p_{ij}\}$) spanned by various possible combinations of service time distributions and routing probabilities. The central task of this framework is to develop mathematical techniques to analytically solve this closed queuing network model. The solution should be able to account for as many service types and as large a workload space as possible, aiming at covering a large design space.

Finally, with regard to the workload, there is a fundamental difference between analytical modeling and simulation/benchmark testing. For the latter, one does not know the actual code path until the testing is over (since there might be conditional branching and dynamic program generation at runtime), whereas for the former, one can assume that the actual code path is known in advance (since the aim of analytic modeling is to try to explain what have happened, i.e., answering "what causes what" and "what if" types of questions).

2.2 Design Space

We want the design space to be as large as possible to encompass as many multicore architectures and workloads as possible. Figure 2.4 depicts such a design space. It is a five dimensional space, including resource-access dimension, thread-scheduling-discipline dimension, program dimension, number-of-thread-per-core dimension, and number-of-core dimension.

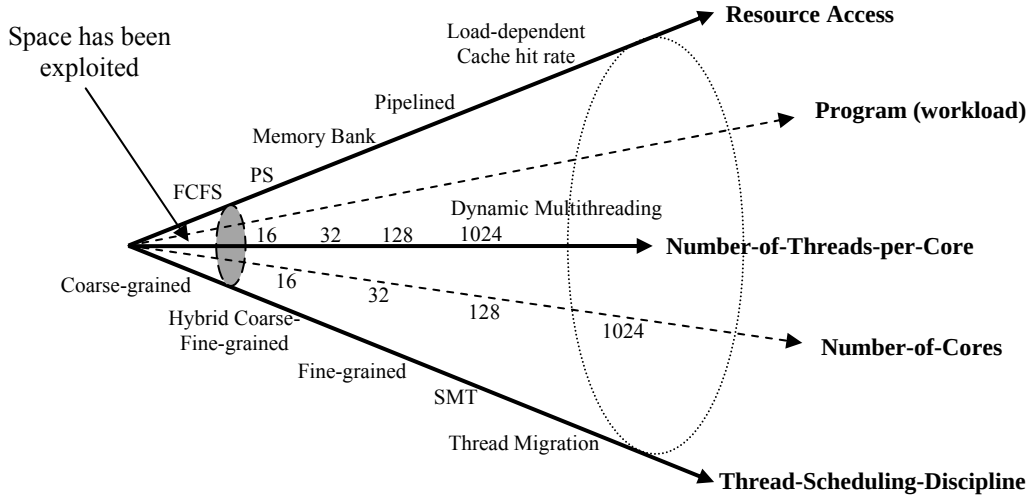

Figure 2.4 Design Space

The Thread-Scheduling-Discipline dimension determines what CPU or core type is in use. The existing commercial processors use fine-grained, coarse-grained, simultaneous multithreading (SMT), and hybrid coarse-and-fine-grained thread scheduling disciplines. Some systems may also allow a thread to be migrated from one core to another.

The Resource-Access dimension determines the thread access mechanisms to MP resources other than CPU. It may include memory, cache, interconnection network, and even a critical region. The typical resource access mechanisms include first-come-first-serve (FCFS), process sharing (parallel access), parallel resources (e.g., memory bank), and pipelined access. For cache access, a cache hit model may have to be incorporated, which may be load dependent.

The Program dimension includes all possible programs. This dimension is mapped to a workload space, involving all possible code paths, for a given type of processor organization (e.g., a code path for a single-core processor with three parallel resources as in Figure 2).

We expect that both Number-of-Cores and Number-of-Threads-per-Core dimensions will reach thousands in the near future. Our theoretical framework needs to be able to deal with MPs of such scale. Moreover, the theoretical framework needs to be able to account for dynamic multithreading, where the number of threads used for a program/program task may change over time.

The queuing network modeling techniques at our disposal restrict the size of the design space to one that must be mathematically tractable. This makes the coverage of the design space in Figure 2.4 a challenge. As shown in Figure 2.4 (i.e., the small cone on the left), the part of the design space that has been (incompletely) explored by the existing work using queuing network modeling techniques is only a tiny part of the entire space.

In what follows, we discuss how our framework allows almost the entire design space in Figure 2.4 to be explored, except dynamic multithreading. We look at different dimensions of the design space separately.

Resource-Access and Thread-Scheduling-Discipline Dimensions: Without resorting to any approximation techniques, the existing queuing network modeling techniques will allow both of these dimensions to be largely explored analytically. Any instance in either of these two dimensions can be approximately modeled using a queuing server model that has local balance equations (i.e., it leads to queuing network solutions of product form or closed form). More specifically, Table 2.1 shows how individual instances in these two dimensions can be modeled by three queuing models with local balance equations, including M/G/∞; M/M/m FCFS (including M/M/1); and M/G/1 PS (Processor Sharing).

Note that the memory banks should be modeled as separate queuing servers and hence, are not listed in this table. Also note that for all the multithread scheduling disciplines except the Hybrid-Fine-and-Coarse-Grained one (to be explained below) in Table 2.1, the service time distribution of a queuing model models the time distribution for a thread to be

serviced at the corresponding queuing server. With these in mind, the following explains the rationales behind the mappings in Table 2.1:

Table 2.1 Component Modeling Using Queuing Models with Local Balance Equations

Component \ Queue Model	M/G/∞	M/M/m FCFS	M/G/1 PS	M/M/1
SMT	√	√		
Fine-Grained Thread scheduling			√	
Coarse-Grained Thread scheduling				√
Hybrid-Fine-and-Coarse-Grained Thread scheduling		√		
Resources dedicated to individual threads	√			
FCFS shared Memory, Cache, Interconnection Network, or Critical Region				√
FCFS Memory with Pipelined Access		√		

- *SMT*: It allows multiple issues in one clock cycle from independent threads, creating multiple virtual CPUs. If the number of threads in use is no greater than the number of issues in one clock cycle, the CPU can be approximately modeled as an M/G/∞ queue, mimicking multiple CPUs handling all the threads in parallel, otherwise, it can be modeled as an M/M/m queue, i.e., not enough virtual CPUs to handle all the threads and some may have to be queued.
- *Fine-grained thread scheduling discipline*: All the threads access the CPU resource will share the CPU resource at the finest granularity, i.e., one instruction per thread in a round-robin fashion. This discipline can be modeled as an M/G/1 PS queue, i.e., all the threads share equal amount of the total CPU resource in parallel.

- *Coarse-Grained thread scheduling discipline*: All the threads access the CPU resource will be serviced in a round-robin fashion and the context is switched only when the thread is stalled, waiting for the return of other resource accesses. This can be approximately modeled as a FCFS queue, e.g., an M/M/1 queue.
- *Hybrid-Fine-and-Coarse-Grained thread scheduling discipline:* It allows up to a given number of threads, say m, to be processed in a fine-grained fashion and the rest be queued in a FCFS queue. This can be modeled as an M/M/m FCFS queue. In this queuing model, the average service time for each thread being serviced is m times longer than the service time if only one thread were being serviced, mimicking fine-grained processor sharing effect.
- *Resources dedicated to individual threads*: Such resources can be collectively modeled as a single M/G/∞ queue, i.e., there is no contention among different threads accessing these resources.
- *FCFS Shared Memory, Cache, Interconnect Network, or Critical Region*: This kind of resources can be approximately modeled as an M/M/1 queue.
- *FCFS Memory with Pipelined Access*: It can be modeled as an M/M/m FCFS queue. The pipeline depth determines how many threads can be serviced simultaneous in the M/M/m FCFS queue.

We note that the resource access dimension also includes load-dependent cache hit rate. The cache hit probability (i.e., the routing probability to move back to the CPU) is generally load-dependent in the sense that it may be either positively or negatively correlated with the number of threads in use due to temporal locality or cache resource contention, respectively. These effects can be accounted for in our framework without approximation, by means of the existing load-dependent routing techniques (e.g. [5]).

We also note that the thread-scheduling-discipline dimension includes thread migration. The thread migration allows a thread to be migrated from one core to another for, e.g., load balancing purpose. This effect can be accounted for without approximation by allowing jobs to have non-zero probabilities to switch from one class to another [4] [6].

More capabilities may be identified and included in these two dimensions as long as they are mathematically tractable.

Program Dimension: In principle, this dimension can be fully explored through a thorough study of the workload space, characterized by the service time distributions and routing probabilities, i.e., a collection of $(\{\mu_i\}, \{p_{ij}\})$'s. However, for the solvable queuing server models in Table 2.1, such as M/M/m and M/M/1 queues, the service time distribution μ_i is a given, i.e., exponential distribution. Since the exponential distribution is characterized by only a single parameter, i.e., the mean service time t_i, it can only capture the first order statistics of the code path segments corresponding to that server, hence providing *a first order approximation of the program dimension or workload space*. Although as part of our future work, we will consider more sophisticated queuing models in an attempt to overcome this limitation, it is widely recognized that the queuing performance for closed queuing networks is insensitive to the service distributions of the queuing servers, generally known as the property of robustness of the closed queuing networks [6]. Hence, we should expect that our first order approximation provides a good coverage of the workload space.

Number-of-Cores and Number-of-Threads-per-Core Dimensions: First, we note that the number of threads dimension should allow dynamic multithreading, meaning that at different program execution stages, the number of active threads may vary. We plan to use a set of ancillary thread classes with different delay loops to join and leave the queuing network modeling a core. It can be easily shown that with n job (or thread) classes and 2^{i-1} threads in the i-th class for $i = 1,\ldots, n$, any number of threads in the range $[1, 2^{n+1}-1]$ can be generated in the core. For example, with $n = 4$, any number of threads in the range of [1, 31] can be

generated. The first thread class has only one thread in it. This thread class runs in the queuing network modeling the core. The rest (n – 1) thread classes run in the delay loops, It can also be shown that by properly setting the delay value for each delay loop, the proposed model can match any distribution of parallelism (i.e., with probability p_k that k threads are presented in the core). The queuing network with these delay loops has closed-form solution. Second, we need to address the scalability issues in calculating the generation functions as the numbers of cores and threads increase. We consider a general closed queuing network modeling an N-core (or core cluster) system with K shared resources. We want to be able to get closed-form generation function G for such closed queuing networks, from which any performance measures can be derived. As long as all the queuing servers in the system have local balance equations (e.g., following the queuing server models in Table 2.1), the generation function (also known as the normalization function in queuing theory) can be generally written as:

$$G = \sum_{\sum_{i=1}^{N+K} m_{i1}=M_1} \cdots \sum_{\sum_{i=1}^{N+K} m_{iN}=M_N} \prod_{j=1}^{N} f_j(m_{jj}) \prod_{j=N+1}^{N+K} f_j(m_{j1}, \ldots, m_{jN}) \quad (2.1)$$

where $f_i(m_{ik})$ is a function corresponding to the probability that there are m_{ik} threads currently in core i (for i = 1,..., N) for thread class k (for k = 1,..., N, i.e., the threads from each core forms a class), $f_j(m_{j1},\ldots,m_{jN})$ is a function corresponding to the probability that there are m_{j1} threads of class one, m_{j2} threads of class two, and so on, in shared memory queuing server j (for j = N+1, …, N+K), and M_i is the total number of threads belonging to core i (for i = 1,..., N). f_i takes different forms for different core organizations, in terms of e.g., CPU, cache, and local memory of different types from the resource-access dimension and thread-scheduling-discipline dimension of the design space.

On one hand, we note that G is defined in the entire design space (with the first order approximation of the program-dimension or workload space). Understanding the general

properties of G over this space will allow the properties of individual points in the design space to be understood, just like function analysis (see Figure 1.1). On the other hand, we also note that the number-of-core and number-of-thread-per-core dimensions create scalability barriers that prevent us from being able to effectively calculate G. This is because the computational complexity for G is O ($N_S M^{N+K}$), where M is the average number of threads per core and N_S is the number of queuing servers per core. Our experiments on an Intel Core-Duo, T2400, 1.83 GHz processor showed that for $N_S = M = 2$ and $K = 1$, it takes about 24 hours to compute the generation function for a 20-core system. Clearly, it is computationally too expensive to cover the entire number-of-core and number-of-thread dimensions. In chapter 4, we develop an iterative procedure to overcome this scalability barrier.

CHAPTER 3

SIMULATION TOOL

In this chapter, we describe our simulation tool for multicore processors based on the thread-level methodology introduced in Chapter 2. In particular, we focus on the simulation model for communication processors (CP), which is the most complex processors in the multicore processor family. The tool applies to other multicore processors as well. Our major goal is to focus on modeling features common to a wide variety of CP architectures and incorporate relevant CP specific features as plug-ins. This tool not only allows user-defined packet arrival processes and code path mixtures to be tested, but also provides a way to allow the maximum sustainable line rate to be quickly estimated. In Section 3.4, Case studies based on a large number of code samples available in IXP1200/2400 workbenches show that the maximum sustainable line rates estimated using our tool are consistently within 6% of cycle-accurate simulation results. Moreover, each simulation run takes only a few seconds to finish on a Pentium 4 computer, which strongly demonstrates the power of this tool for fast CP performance testing.

3.1 Design Ideas

The following three key ideas underlay the CP simulation tool development.

Focus on emulating common CP features while taking the relevant performance impacts of CP-specific features into account through user provided models. Our approach attempts to strike a balance between complexity and simplicity by adopting a hybrid simulation-and-modeling based approach. Specifically, our approach focuses on faithfully emulating important features common to a wide variety of CP architectures (e.g., multithreading and multi-

core) and account for the relevant performance impacts of CP-specific features (e.g., I/O interface, cache, memory, memory controller, and bus architectures) through user provided models, called plug-ins. For example, by focusing on the throughput, delay, and loss performance, our tool only requires a plug-in that captures the memory access latency for each CP memory. In other words, the plug-in only needs to capture the delay performance aspect of memory accesses, which can be modeled by the user, based on, for example, a queuing model or even an empirical chart, without having to emulate the processing details cycle-by-cycle. This design approach makes the simulation tool generic and adaptable to a wide range of CP architectures with the addition of a set of user provided plug-ins, capturing CP architecture specific features. With limited number of I/O and memory interfaces for CPs in general, the number of plug-ins needed are generally small, e.g., less than a dozen.

Capture important events only. The existing approach which attempts to accurately pin down the CP performance generally resorts to cycle-by-cycle or instruction-level simulation. This not only makes the simulation slow and storage space demanding, but also requires the availability of the executable program as input for the simulation. What is available to us, however, is only a piece of pseudo code for packet processing tasks mapped to each core, which defies the use of cycle-by-cycle simulation. Instead, we adopt the thread-level performance analysis methodology introduced in Chapter 2. Namely, we propose to identify a sequence of events that may have significant impact on the thread-level performance, identifiable from the pseudo code and perform event-by-event, rather than instruction-by-instruction simulation. Since the number of important events identifiable in each code path is generally small, e.g., one dozen to a few dozen, one can expect that event-by-event simulation would be significantly faster than cycle-by-cycle simulation. Identifying important events is not difficult. For throughput, delay, and loss performance analysis, the important events may include memory and I/O accesses that results in a context switching, cache accesses that may cause a context switching, events that cause serialization effects, such as a critical section, and events

that cause run-time code generation, such as packet fragmentation for which the code size is a function of the packet size. Although the instruction level activities, such as per instruction cycle time and instruction-level-pipelining (ILP) aborts, cannot be directly captured based on a piece of pseudo code, the average impact of these activities on the overall throughput, delay, and loss performance may be modeled. For example, by defining an event for code branching and associating with each branching event an average ILP abort cost in terms of wasted cycles, the impact of ILP aborts can be accounted for on average. In fact, using average data to simplify the simulation is not original. For example, the average per instruction cycle time has been widely used in processor performance analysis.

In summary, combining the event-by-event simulation and the separation of CP common features from CP specific features makes the tool very lightweight in terms of both time and space complexities. Moreover, simulation at the event level makes it possible to allow the use of pseudo code, rather than an exact program, as input for the simulation.

Allow for sustainable line rate estimation. No matter how lightweight a simulation tool would be, the simulation time is guaranteed to be prohibitively long if the goal is to perform exhaustive statistic analysis. This is because there are virtually unlimited numbers of possible packet arrival processes and mixtures of code paths the threads in each core may concurrently handle, which has made a CP the most difficult one to simulate, among all MPs in the MP family. Unless a user has in mind small numbers of targeted packet arrival processes and code path mixtures to be tested, performing exhaustive statistical analysis in a large design space is guaranteed to be extremely time consuming, if possible. In practice, for most CP programmers and designers, what they really want to know is, for a given task-to-CP-topology mapping, whether the CP can sustain wire-speed forwarding performance or not. Unfortunately, traditionally, the data inputs, including packet arrival processes and code path mixtures, are provided by the user of the tool, rather than part of the tool design. As a result, all the existing CP simulation tools were developed without being concerned with how the input data should be

generated. This makes it difficult for a user or designer to effectively use such a tool to test whether the CP can keep up with the line rate or not. To address this issue, as part of the tool design, we develop a systematic approach to allow the sustainable line rate to be estimated for any given task-to-CP-topology mapping. In this approach, the user does not have to provide any packet arrival processes, nor code mixtures as input to the tool, but simply a piece of pseudo code for the tasks mapped to each core. The tool will automatically return the line rate the CP can sustain, under such a mapping.

3.2 Simulation Architecture

In this Section, we introduce the generic CP organization, the code path definition, the simulation model, and an approach to estimate the sustainable line rate.

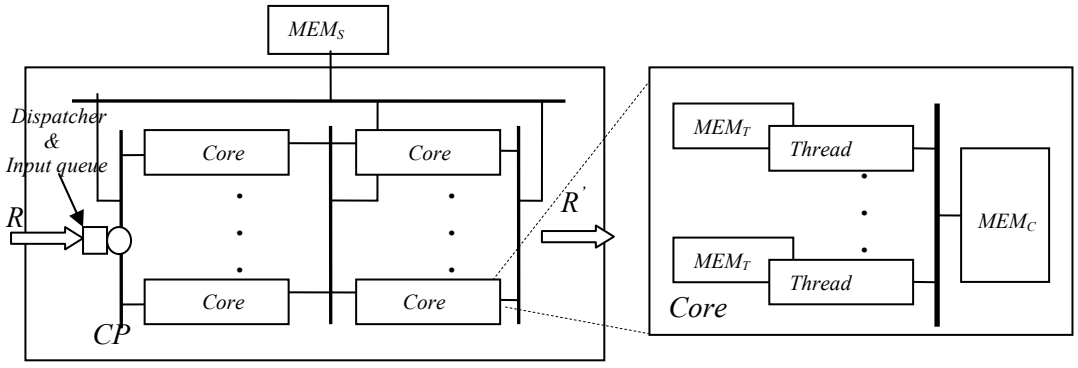

Figure 3.1 Generic CP Organization

Generic CP Organization: Based on the aforementioned methodology, we consider a generic CP organization depicted in Figure 3.1. This organization focuses on the characterization of multicore and multithreading features common to most of the CP architectures, leaving all other components being summarized in highly abstract forms. More specifically, in this organization, a CP is viewed generically as composed of a set of cores and a set of on-chip or off-chip supporting components, such as I/O interfaces, memory, level one and

level two caches, special processing units, scratch pads, embedded CPUs, and coprocessors. These supporting components may appear at three different levels, i.e., the thread, core, and system (including core cluster) levels, collectively denoted as MEM_T, MEM_C, and MEM_S, respectively. Each core supports multiple threads which are scheduled based on a given thread scheduling discipline. Cores may be configured in parallel and/or multi-stage pipeline (a two-stage configuration is shown in Figure 3.1). Packet processing tasks are partitioned and mapped to different cores at different pipeline stages or different cores at a given stage. A dispatcher distributes the incoming packets to different core pipelines based on any given policies. Backlogged packets are temporarily stored in an input buffer. A small buffer may also present between any two consecutive pipeline stages to hold backlogged packets temporarily. Packet loss may occur when any of these buffers overflow. The tool is concerned with the CP throughput, latency, and loss performance only and the power and memory resource constraints are assumed to be met. This implies that we do not have to keep track of memory or program store resource availabilities or power budgets.

Code Path: The code path concept was introduced in Chapter 2. Here we give a more accurate definition of it. A code path is defined at the core level. For tasks mapped to a given core, a piece of pseudo code for these tasks can be written. Then a unique branch from the root to a given leaf in the pseudo code is defined as a code path associated with that core. An incoming packet to the core is accepted if there is a free thread in the core, and is associated with one code path, or a sequence of instructions that the core needs to execute throughout the life-time that the packet is in that core.

In this simulation model, a code path is broken down into a sequence of segments of instructions intermediated by events. For each segment, we are only concerned with the segment length, i.e., the number of instructions in the segment (which can be easily estimated on the basis of the pseudo code), or more precisely, the number of core cycles the core

arithmetic logical unit (ALU) has to spend on the segment, assuming the average per instruction cycle time is known. Hence, a code path can be formally defined as follows:

$T_k(M_k; m_{1,k}, t_{1,k}, \tau_{1,k}, \ldots, m_{Mk,k}, t_{Mk,k}, \tau_{Mk,k})$: Code path k with event $m_{i,k}$ occurred at the $t_{i,k}$-th core clock cycle and with event duration $\tau_{i,k}$, where $k = 1, \ldots, K$ and $i = 1, 2, \ldots, M_k$, where K is the total number of code paths in the pseudo code mapped to the core and M_k is the total number of events in the code path.

$|T_k|$: the code path length or the total number of core clock cycles in the code path $T_k(M_k; m_{1,k}, t_{1,k}, \tau_{1,k}, \ldots, m_{Mk,k}, t_{Mk,k}, \tau_{Mk,k})$, where $k = 1, 2, \ldots, K$.

A graphical representation of such a code path is given in Figure 3.2.

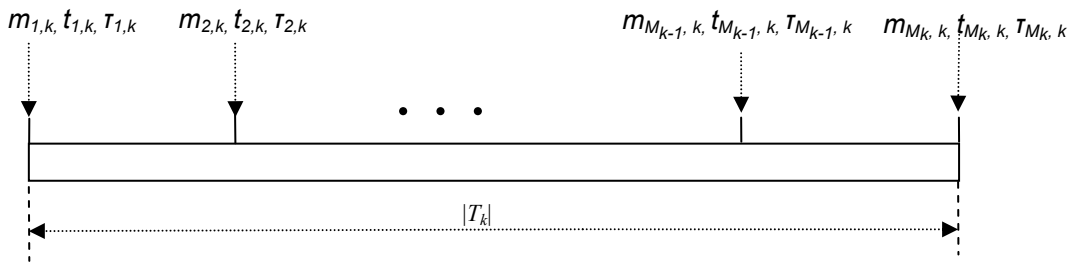

Figure 3.2 A graphical Representation of A Code Path

We note that a code path thus defined is simply a sequence of events with event inter-arrival times ($t_{i+1,k} - t_{i,k}$) for $i = 1, 2, \ldots, M_k - 1$. We also note that the first instruction and the last instruction in the code path must be treated as events. For these events, $\tau_{i,k} = 0$. For an event $m_{i,k} \in MEM_T$, MEM_C, or MEM_S, $\tau_{i,k}$ represents the loaded resource access latency. To account for the serialization effect caused by, for example, a critical section, two events must be included, indicating the start and end of the critical section. Again, for these events, $\tau_{i,k} = 0$.

An important event is defined as one that is expected to have an impact on the throughput, delay, and loss performance. Currently, we have defined the following four types of important events: (1) events for the start and end of the code path; (2) resource access events

which may cause significant delay and thread level interactions (context switching), events $m_{i,k} \in$ MEM_T, MEM_C, or MEM_S ; (3) events that cause a serialization effect; and (4) events that cause dynamic code generation. More types of events can be incorporated if they are expected to contribute significantly to the throughput, delay, and loss performance. For example, a new type of event that identifies branching points in the code can be included for the purpose of estimation of ILP abort cost caused by branching, if the abort effect cannot be neglected.

Simulation Model: Our simulation tool focuses on three performance measures: throughput, delay, and loss. All three measures can be obtained at runtime as long as the latency L_k, a packet with code path k in each core can be simulated, which can be expressed conceptually as:

$$L_k = |T_k| + \sum_{j=1:Mk} (\tau_{j,k} + \tau^w_{j,k}), \quad (3.1)$$

where $\tau^w_{j,k}$ is the thread waiting time in the ready state after the event $m_{j\,k}$ finishes and $\tau^w_{j,k}$= 0 if event $m_{j,k}$ does not cause a context switching. For $m_{i,k} \in$ MEM_T, MEM_C, or MEM_S, $\tau_{j,k}$ is dependent on the nature of $m_{j,k}$ access (number of memory reads or writes), the access speed (bus speed and memory speed), and access contention resolution mechanisms, such as the memory access pipelining and queuing architectures, which must be estimated based on a user provided plug-in. $\tau^w_{j,k}$ is dominated by multithreading effects, which is the core parameter to be simulated at runtime. $|T_k|$ is the total number of core clock cycles the core ALU spends on the packet.

Hence, for throughput, delay, and loss performance analysis, in general, all we need from the user is a set of plug-ins that estimate $\tau_{j,k}$ for $m_{i,k} \in$ MEM_T, MEM_C, or MEM_S at runtime. Note that, although modeling CP-specific features in general is a nontrivial task, it should not be difficult to come up with empirical memory access latency models, e.g., in the form of charts or tables for a given CP. For example, by loading a given memory with different number and types

of requests and measuring the corresponding loaded latencies using a cycle-accurate simulator or test board, one can build empirical charts or tables offline to be used to quickly estimate the memory access latency at runtime. There is no need to emulate the microscopic process for memory access at run-time, saving significant simulation time. As we shall see in the next section, with the unloaded memory access latencies provided by Intel, as well as a memory access waiting time estimated based on a simple FIFO queuing model, our simulation tool accurately characterizes IXP1200/2400 performance without further information about IXP1200/2400 specific features.

With the previously described preparation, now we describe our simulation model, which focuses on emulating non-CP specific components, including core topology, multithreading, code path, code path mixtures, and packet arrival processes, pertaining to all the CP architectures, with a limited number of plug-ins for resource access latency estimation. These plug-ins are pre-developed and plugged into the simulation model. Figure 3.3 gives a logic diagram for the proposed simulation model, which is composed of four major components: (1) a simulation core based on the generic CP organization described in Figure 3.1; (2) code path association with a packet in a core; (3) a packet arrival process; and (4) a set of plug-ins to the simulation core.

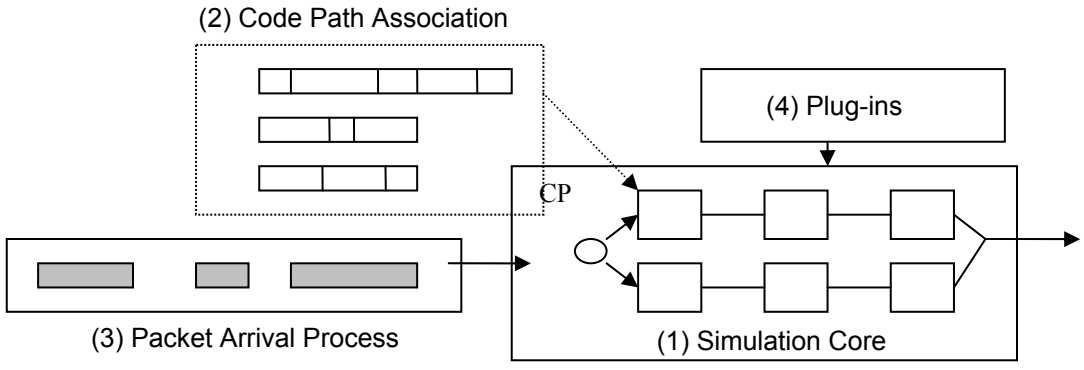

Figure 3.3 CP Simulation Model

Based on the generic CP architecture in Figure 3.1, the simulation model focuses on emulating multithreaded cores which can be configured in any pipeline/parallel topology. Each core is modeled at a highly abstract level, running any number of threads based on a given thread scheduling discipline (our current design includes fine-grained, coarse-grained, and TDM-based disciplines). No further details of the core are modeled. A thread in a core that receives a packet will be assigned a code path. The way to assign code paths to threads in a core determines the mixture of code paths in that core. The packet arrival process can be generated from real traces (which also determine the code path mixture in each core), stochastic models, or deterministic models. Traditionally, the code path assignment and packet arrival process generation are not part of the simulation tool design, but as user provided inputs. Since all four components can be designed independent of one another, the design of components (1) and (4) combined constitutes a fast performance analysis tool in the traditional sense. In other words, as in traditional approaches, our tool allows any user provided packet arrival processes and/or mixtures of code paths to be simulated. Our goal, however, is to also design components (2) and (3) (to be discussed in Section 3.3) such that for any given task-to-CP-topology mapping, the tool can quickly return the maximum line rate the CP can sustain.

With the CP organization in Figure 3.1 and the event annotated code path in Figure 3.2, the fast simulation tool is developed based on the event-driven simulation approach. To help understand why such a tool can be made to execute quickly, here we give an intuitive explanation by way of a simple example. Consider a code path in Figure 3.4.

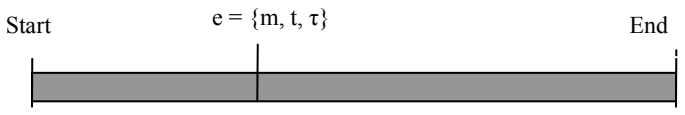

Figure 3.4 An Example of Event-Annotated Code Path

In this code path, there are only three events, the start, end, and *e*. Event e takes place at the *t*-th cycle for resource, *m,* access with loaded latency *τ*. Now consider two threads in a core, each handling a code path as in Figure 3.4. They share the ALU resource based on a fine-grained thread scheduling discipline (i.e., switch context at every instruction). Figure 3.5 gives the instruction execution timeline for the two code paths. The dark gray parts represent the code path segments. The light gray parts represent the cycles spent on event *e*, i.e., the loaded *m* access latencies. The white part stands for the cycles spent in the ready state waiting for execution after event *e* finishes. In this case, each code path involves three event boundaries: the start of the code path, the end of the code path, and the start and end of event *e*. The arrows represent the switches of control from one thread to the other after executing one instruction. The idea is not to simulate each and every switch of control, but only the cycles at the event boundaries, i.e., the positions indicated by vertical lines. Since each code path may have up to a few dozens of events, only a few dozens of event boundaries need to be simulated per packet. As a result, the event-driven simulation tool that captures only those events can run several orders of magnitude faster than cycle-accurate simulation tools, as our testing results showed (a few seconds per simulation run on a Pentium 4 Computer).

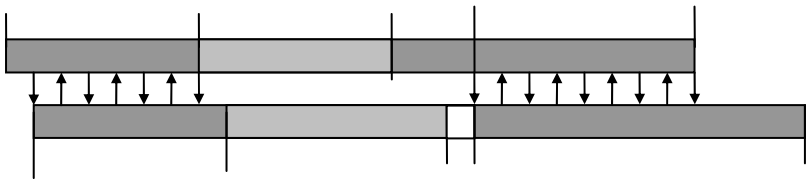

Figure 3.5 Event-Level Simulation

3.3 Sustainable Line Rate Estimation

The simulation model proposed in the previous section allows any packet arrival processes and code path mixtures to be simulated. In this section, we go one step further by designing the packet arrival processes and the code path mixtures to allow the maximum line rate a CP can sustain to be quickly simulated.

In the router industry, the performance of a router is judged mainly by whether its network interface cards can sustain wire-speed forwarding performance or not. A typical testing scenario is to use back-to-back, minimum-sized packets running at the line rate as input for the testing and all the packets are loaded with a typical code path, such as the code path that carries out basic tasks for IP forwarding. Following this industry practice, we would have already achieved our objective. However, while the packet arrival process thus generated makes sense, the use of a "typical" code path to determine whether a CP can sustain wire-speed forwarding performance may not always be a good idea, as long as untypical code paths may occur with non-negligible probabilities. For this reason, we adopt the industry practice on packet arrival process, but design our own code path assignment mechanism. In what follows, we discuss these two aspects separately.

Packet Arrival Process: Denote R as the line rate. Then the minimum packet time is P/R, where P is the size of a minimum-sized packet. We define a deterministic packet arrival process as minimum-sized packet arriving at fixed packet time interval T_P. For this process, the packet arrival rate $r = P/T_P$. The r value at which the packet loss is about to occur is then the maximum line rate the CP can sustain and if $r \leq R$, we say that the line rate R can be sustained. Here, what code path mixture should be used as input to the simulation is yet to be specified, discussed below.

Code Path Assignment: Assume the user of the tool does not have a typical code path mixture in mind to test whether the line rate can be sustained. Our goal is then to identify the worst-case mixture of code paths that gives the lowest line rate the CP can sustain. This will ensure that the estimated sustainable line rate can always be achieved, under any mixtures of code paths. To this end, we first make the following two important observations.

First, intuitively, it is clear that the worst-case scenario for a given core will occur only when all the code paths in a code path mixture are the same. This is because having one code path for all the threads in the core will stress a particular resource the most. For example, if the

longest code path may potentially create a bottleneck for the core ALU resource, then loading the core with this code path for all the threads will stress the core ALU resource the most. This observation significantly reduces the complexity for the identification of the worst-case mixture of code paths. In this case, one may then test different potential worst-case code paths separately, rather than different combinations of code paths, significantly reducing the number of test cases.

Second, with a deterministic arrival process and code path mixture, and at the saturated arrival rate, queuing cannot help improve the throughput performance simply because there is no traffic fluctuation to provide rooms for the buffers between pipeline stages to offload the queued packets once the queue levels build up. In this case, we may view a core pipeline as a buffer-free system and the throughput for a core pipeline is determined by the throughput at the bottleneck pipeline stage.

Based on these observations, we can logically decompose the problem into two sub-problems: (1) identifying the bottleneck pipeline stage; and (2) finding the worst-case code path that leads to the minimum throughput for the core at the bottleneck stage. In practice, these two sub-problems may not be separable. In what follows, we present a mechanism to address these problems.

In principle, the worst-case code path for one core must be associated with some given mixtures of code paths (not necessarily the worst-case mixtures of code paths) for other cores. The reason is that packet processing processes in different cores are coupled together through the sharing of supporting components of type MEM_S (see Figure 3.1). As a result, to identify the worst-case code path for a particular core, one must also identify the associated code path mixtures for other cores. These code path mixtures are then associated with the arriving packets as input to various cores that simulates the entire system (see Figure 3.1) as a whole. This simulation run will return the sustainable line rate for that core and this process must be repeated for all the cores to identify the bottleneck core. However, in practice, this coupling

effect is not strong for the following reasons. First, different cores in different pipeline stages generally carry out distinct packet processing tasks. For example, in a three stage pipeline, the first stage may perform initial processing of incoming packets by loading the packets into an external DRAM. The second stage may perform major packet processing functions, which may involve significant table lookups in an external SDRAM. The last stage may mainly provide queue scheduling functions. As a result, the chances for different stages to interact with one another through MEM_S types of resource accesses are small. Second, due to the wide use of multiple external memory interfaces, memory banks, and core clusters sharing different L2 caches in CP design, such coupling effects are further reduced. Hence, in our current tool design, we simply overlook such coupling effects. Nevertheless, our tool can be easily extended to take such effects into account, e.g., by modeling the MEM_S accesses from other cores as a background random process derived from the *core*-access-intensive code path loaded to other cores.

Without considering the inter-core coupling effects, the problem is then reduced to one of finding the worst-case code paths for individual cores, separately. The worst-case code path k by definition generates the largest core latency L_k (see Eq. (3.1)). However, since without simulation, we have no idea about the values for the last term in L_k. The approach taken is to simply neglect $\tau^w_{j,k}$ and approximate L_k by $|T_k| + \sum_{j=1:\,Mk} \tau_{j,k}$, where $\tau_{j,k}$ is now the unloaded latency. This approximate latency can be estimated easily for each and every code path. We simply select an x % of the code paths with the largest L_k values to be considered as potential worst-case code paths to be tested. The reason to choose x % rather than just the one with the largest approximated L_k to be tested is simply to compensate for the inaccuracy of such estimations.

An initial testing of the above approach is encouraging. We tested the above approach against 50 randomly generated pseudo codes with each having 100 to 1000 branches or code paths with multiple MEM_T, MEM_C, and MEM_S types of memory accesses. Since the tool finishes

running each case within 10 seconds on a Pentium IV PC, an exhaustive search of the worst-case code path for each pseudo code was performed. The worst-case code path thus found is then compared against the ones found by the above approach. For all 50 cases, we find that the true worst-case code path always falls into the top 1% (i.e., x = 1) of the entire code paths pool. This means that even for a pseudo code with up to 1000 code paths, only ten simulation runs with a fixed T_P are needed to identify the true worst-case code path, which takes a bit more than one minute. For a CP with a dozen of cores, this will take only dozens of minutes to pin down the worst-case code paths for all the cores.

Finally, for each core loaded with the worst-case code path, several simulation runs with different T_p values are performed to identify the maximum sustainable line rate for that core. This process is repeated for all the cores to identify the bottleneck pipeline stage for each core pipeline and finally the maximum sustainable line rate for the entire CP.

3.4 Simulation Testing

In this section, the accuracy of the proposed tool is tested against the CAS (Cycle-Accurate Simulators), i.e., IXP 1200/2400 SDK Developer workbenches [8] [9]. With a set of code samples available in both IXP1200/2400, the sustainable line rates obtained from our tool are compared with those from CAS. For all the code samples, there are only a few number of code paths for each core and we can afford to perform exhaustive search for the bottleneck core and the corresponding worst-case code path. For this reason, the simulation focuses on testing the accuracy of the simulation tool only, assuming the bottleneck core and the corresponding worst-case code path are known. The code samples and corresponding simulation setups are described in Section 3.4.1 and Section 3.4.2 presents the test results

3.4.1. Simulation Setup

Since all the cores in IXP1200/2400 run a coarse-grained thread scheduling discipline, our simulation tool is configured to run the coarse-grained thread scheduling algorithm as well.

The functions in the core topology for IXP1200 and IXP2400 sample applications are briefly described as follows.

IXP1200 code samples: Four different code samples, *Packet Count* [1], *Generic IPv4 Forwarding*, *Layer-2 Filtering*, and *ATM/Ethernet IP Forwarding*, available in IXP1200 Developer workbench [8] are tested. The worst-case code paths at the bottleneck cores for these code samples are given in Appendix A. The complete implementation details can be found in the Intel IXP1200 building blocks application design guide with the Developer workbench. In the following description of code samples, we focus on the functions mapped to the bottleneck core.

*Packet Count***:** this code sample counts the number of packets received. A receive thread checks for data on the MAC port, transfers packet from MAC port to receive buffers. After packet reception is complete, the thread moves the packet into SDRAM and reads the packet header into the core. A counter is maintained in SCRATCH and is incremented on receiving a packet.

*Generic IPv4 Forwarding***:** after packet reception as in *Packet Count*, RFC1812 generic IPv4 forwarding is implemented in this code sample.

*ATM/Ethernet IP Forwarding***:** This code sample is a mixed code implementation of ATM /Ethernet IP forwarding. Only Ethernet-to-ATM flow is considered in the test. The header checksum check, TTL update, and IP lookup are performed in the receive block after packet reception as in *Packet Count*. Then the LLC/SNAP and modified IP headers are written back into the SDRAM. When the frame fragment with EOP (End of Packet) information is received, AAL5 trailer information is written into the SDRAM buffer and the complete PDU is enqueued for CRC generation at the next pipeline stage.

*Layer-2 filtering***:** This code example implements Ethernet protocol, MAC address filtering and layer 2 forwarding in the receive block after packets are received.

Packet Count, *Generic IPv4 Forwarding*, and *Layer-2 Filtering* code samples are mapped to two core pipelined stages as shown in Figure 3.6 and ATM/Ethernet Forwarding is

mapped to three core pipeline stages as shown in Figure 3.7. The original code samples are modified to allow only one core at the receive stage handling packets coming from a single port. As a result, the receive core becomes the bottleneck core to be tested. The code samples can also be changed to allow configuration of the number of threads from one to four.

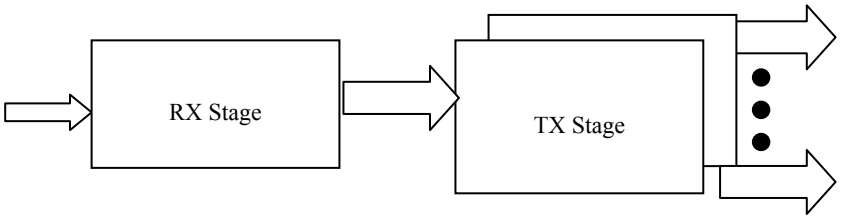

Figure 3.6 Pipeline Configuration for Packet Count, Generic IP Forwarding and Layer-2 Filtering

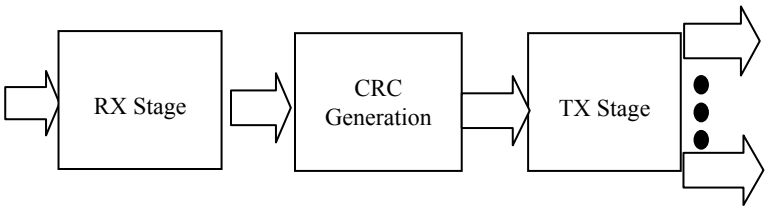

Figure 3.7 Pipeline Configuration for ATM/Ethernet Forwarding

Packet Count, *Generic IPv4 Forwarding*, and *Layer-2 Filtering* code samples are mapped to two core pipelined stages as shown in Figure 3.6 and ATM/Ethernet Forwarding is mapped to three core pipeline stages as shown in Figure 3.7. The original code samples are modified to allow only one core at the receive stage handling packets coming from a single port. As a result, the receive core becomes the bottleneck core to be tested. The code samples can also be changed to allow configuration of the number of threads from one to four.

IXP2400 code samples: *IPv4 Ethernet*, *DiffServ POS*, and *MPLS* in IXP2400 Developer workbench [9] are tested. Again, the associated code paths are given in Appendix A.

In what follows, the code samples are briefly explained. All of these applications have five blocks and the same pipeline configuration as shown in Figure 3.8. The complete implementation details can be found in the Intel IXP 2400 building blocks application design guide with the Developer workbench.

The first block is a packet receive block and uses a scratch ring to communicate with the next block. The second block is a functional block where application specific functional pipeline executes in parallel on four cores. The pipeline has five microblocks in *DiffServ POS* application: PPP decapsulation/classify microblock, 6-tuple classifier microblock, TCM meter microblock (ignored for our simulation setup: in case of no traffic profile in effect, packets may only pass through a classifier and a marker [RFC 2475]), DSCP Marker microblock and IPv4 forwarder microblock. In *MPLS* and *IPv4 Ethernet* applications, the pipeline consists of two microblocks: MPLS processing and IPv4 forwarder microblocks for *MPLS* and Ethernet decapsulation/classify/filter and IPv4 forwarder microblocks for *IPv4 Ethernet*. The third block is the queue manager which performs enqueue/dequeue operations on the hardware-assisted SRAM queues. The queue manager receives enqueue requests from the functional pipeline through a scratch ring. Another scratch ring is fed with dequeue requests from the CSIX scheduler. The fourth block is the CSIX scheduler which selects constant-length packet segments to be transmitted to the CSIX fabric. The final block is the CSIX transmit block which receives transmit messages from the queue manager and moves packet segments into a transmit buffer.

For all the IXP2400 code samples we tested, the number of threads in use is not configurable in the original code samples, which is fixed at eight. To test the functional block, three of the four cores are disabled and the remaining core creates a bottleneck at this block.

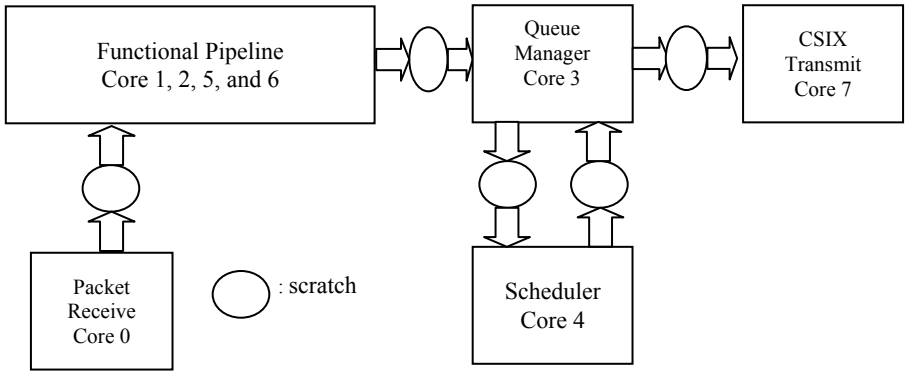

Figure 3.8 Ingress Blocks for IXP2400 Code Samples

Our simulation tool only needs to run a single core, corresponding to the bottleneck core for sample applications described above. The sustainable line rate for this bottleneck core is compared with that of CAS simulation involving the entire multi-stage pipeline. All the worst-case code paths for the corresponding bottleneck cores in table format are listed in Appendix A. The first column lists the task performed in each code path segment; the second column gives the segment length in terms of core clock cycles; the third column describes the type of resource accesses between segments; and the last column gives the unloaded resource access latency for each resource access. We assume that in the presence of resource access contentions, the resource access requests will be serviced based on a simple FIFO queuing mechanism. This means that unloaded resource access latencies and a set of simple resource access FIFO queues are the only IXP1200/2400 specific features or plug-ins used in our simulation tool. The rest are generic or common features pertaining to all the CP architectures. This indicates that our simulation tool is indeed generic and easily adaptable to a specific CP architecture. Clearly, the code paths as given in Appendix A can be easily derived from a piece of pseudo code provided by the user.

The parameter settings for the simulation are as follows:

IXP1200/2400: ME clock rate = 200/600 MHz, Packet size = 64/64 bytes,

SDRAM = 24/64MB, SRAM = 1/64 MB (for each channel of two).

3.4.2. Test Results

In this Section, we compare sustainable line rates obtained from our tool with those obtained from IXP1200/2400 CAS for the code samples described in Section 3.4.1. Tables 3.1 through 3.4 give the results for different cases in similar formats. For IXP 1200 case studies in Tables 3.1 through 3.4, the first column gives the number of threads configured; the second and the third columns list the core latencies and the sustainable line rates obtained from our tool and CAS, respectively. The last column lists the percentage difference between the two sets of results. The table format for IXP2400 case studies in Table 3.5 is similar to the ones for IXP 1200 case studies, except the first column, where now the applications are listed, rather than the number of threads, which for IXP2400 is fixed at eight. As one can see, for all the cases studied, the results obtained from our tool are within 6% of the CAS results. Moreover, each simulation run finishes in a few seconds on a Pentium 4 Computer. Since there are significant differences between IXP1200 and IXP2400 architectures, such consistent agreement of the two provides strong evidence, indicating that our tool can serve as an effective tool to aid the initial programming of a CP as well as new CP architecture design.

Finally, we note that in all the code samples, there is a critical section in the receive stage. In particular, the critical section dominates the code path of the *Packet Count* sample, constituting about 67% of the total code path, from the first task to the 6^{th} task in the code path for Packet Count Sample given in Appendix A. This dominant serialization effect causes the line rate to decrease when more than two threads are configured as shown in Table 3.1. This effect is successfully captured by our tool, which accounts for the impact of the critical section by recognizing the start and end of critical section events in the code path. Interested readers may

refer to [10] for detailed explanation on why this critical section causes the decrease of line rate when adding more than two threads.

Table 3.1 Tool versus CAS (IXP1200) for *Packet Counting*

Threads	Tool		CAS		% Error rate |R1-R2|*100/R2
	Total Latency(TL) (cycles)	Receive rate R1 (Mbps)	TL	R2	
1	296	334	293	337	0.89
2	375	525	380	518	1.40
3	645	460	627	473	2.74
4	875	450	856	460	2.17

Table 3.2 Tool versus CAS (IXP1200) for *Generic IPv4 Forwarding*

Threads	Tool		CAS		% Error rate
	TL	R1	TL	R2	
1	560	183	537	183	0.00
2	590	347	600	328	5.79
3	687	447	687	431	3.71
4	936	438	876	449	2.45

Table 3.3 Tool versus CAS (IXP1200) for *ATM/Ethernet IP Forwarding*

Threads	Tool		CAS		% Error rate
	TL	R1	TL	R2	
1	732	140	724	140	0.00
2	830	247	812	250	1.20
3	985	312	981	311	0.32
4	1260	326	1184	343	4.96

Table 3.4 Tool versus CAS (IXP1200)
for *Layer-2 Filtering*

Threads	Tool		CAS		% Error rate
	TL	R1	TL	R2	
1	735	140	730	140	0.00
2	816	251	798	257	2.33
3	960	320	978	314	1.91
4	1168	351	1304	354	0.85

Table 3.5 Tool versus CAS (IXP2400)
with 8 Threads

Applications	Tool		CAS		% Error rate
	TL	R1	TL	R2	
Diffserv Pos	2960	830	2950	832	0.24
MPLS	2824	870	2750	890	2.25
IPv4 Ethernet	2898	849	2935	839	1.19

3.5 Related Work

There are a vast number of processor simulation tools available [1] [12-26] (e.g. cycle-accurate, allowing detailed timing analysis, and providing primitives for flexible component modeling). Particularly relevant to our work are the Network Processor analysis tools (e.g., [1] [16-24]).

Most CP simulation software (e.g., [16-19]) aims at providing rich features to allow detailed statistical or per-packet analysis, which is useful for program fine tuning, rather than fast CP performance testing. Even for the most lightweight CP simulator described in [16], it is reported that to simulate one second of hardware execution, it takes 1 hour on a Pentium III 733 PC. Moreover, it assumes the availability of the executable program or microcode as input for the simulation. On the other hand, the algorithms for data path functions to CP core topology mapping (e.g., [21] [23] [24]) are generally fast, but at the expense of having to overlook many essential processing details that may have an impact on the overall system performance. To make the problem tractable, a common technique used in these approaches is to partition the

data path functions into tasks and each task is then associated with one or multiple known resource demand metrics, e.g., the core latency and program size. Then an optimization problem under the demand constraints is formulated and solved to find a feasible/optimal mapping of those tasks to a pipelined/parallel core topology. Since the actual resource demand metrics for each task are, in general, complex functions of mapping itself, and are a strong function of the number of threads and thread scheduling discipline in use at each core, these approaches cannot provide mappings with high accuracy. Although the approach in [23] accounts for certain multithreading effect, it only works for a single memory access and under a coarse-grained scheduling discipline.

From previous work, all the above existing tools lack of the following feature needed to address the problem at hand. Namely, they are not concerned with how the packet input process should be generated, which is assumed to be provided by the user, rather than part of the tool design. Given that there are virtually unlimited numbers of possible packet arrival processes and mixtures of packets carrying various code paths, it is a daunting task for a user of an existing tool to decide what input processes should be tested, or what statistics should be collected. Note that no matter how fast a simulation tool may be, the simulation time is guaranteed to be prohibitively long if the goal is to perform exhaustive statistical analysis, based on the simulation data collected from a large number of packet arrival processes and mixtures of code paths

CHAPTER 4

A THEORETICAL FAMEWORK FOR DESIGN SPACE EXPLORATION

In this chapter, we develop a theoretical framework for a large design space based on the methodology introduced in Chapter 2. Section 4.1 introduces the queuing network model that covers the entire design space in Figure 2.4. Section 4.2 proposes an iteration algorithm that addresses the scalability issue concerning the queuing network modeling technique, as explained in Section 2.2. Section 4.3 provides the testing results in a large design space for the proposed analytic model versus simulation results using the simulation tool developed in Chapter 3. Finally, Section 4.4 discusses the related work.

4.1 Queuing Network Model

For MPs in the design space covered by the queuing network models in chapter 2, all the performance measures can be derived from a generation function, which is described mathematically as follows.

First, we define *N* as the total number of jobs (or threads) for the entire system and it follows that,

$$N = \sum_{i=1}^{M} k_i, \qquad k_i = \sum_{r=1}^{R} k_{ir}\ , \tag{4.1}$$

where k_{ir} is the number of jobs in the *r*th job class at the node *i* and *M* is the total number of queuing servers and *R* is the number of job classes in the system.

According to the BCMP theorem [27], the state probabilities of the system can be written as:

$$\pi(\mathbf{S}_1, \dots, \mathbf{S}_M) = \frac{1}{\mathrm{G}(\mathbf{N})} \prod_{i=1}^{M} f_i(\mathbf{S}_i), \tag{4.2}$$

where the state of the *i*th node is $\mathbf{S}_i = (k_{i1}, .., k_{iR})$ and the population vector containing the total number of jobs is $\mathbf{N} = \sum_{i=1}^{M} \mathbf{S}_i$ and $\mathrm{G}(\mathbf{N})$ is the so-called normalization constant or generation function of the system and it is given by:

$$\mathrm{G}(\mathbf{N}) = \sum_{\sum_{i=1}^{M} \mathbf{S}_i = \mathbf{N}} \prod_{i=1}^{M} f_i(\mathbf{S}_i), \tag{4.3}$$

The $f_i(\mathbf{S}_i)$'s are the relative state probabilities of the state $\mathbf{S}_i$ at the node *i* , which are defined as follows:

$$f_i(\mathbf{S}_i) = \begin{cases} k_i! \dfrac{1}{\beta_i(k_i)} \cdot \left(\dfrac{1}{\mu_i}\right)^{k_i} \cdot \displaystyle\prod_{r=1}^{R} \frac{1}{k_{ir}!} e_{ir}^{k_{ir}}, & \text{for} -/\ \mathrm{M/m} - \mathrm{FCFS}, \\ k_i! \displaystyle\prod_{r=1}^{R} \frac{1}{k_{ir}!} \cdot \left(\frac{e_{ir}}{\mu_{ir}}\right)^{k_{ir}}, & \text{for} -/\mathrm{G}/1 - \mathrm{PS\ \&\ LCFS\ PR}, \\ \displaystyle\prod_{r=1}^{R} \frac{1}{k_{ir}!} \cdot \left(\frac{e_{ir}}{\mu_{ir}}\right)^{k_{ir}}, & \text{for} -/\ \mathrm{G}/\infty\ (\mathrm{IS}), \end{cases} \tag{4.4}$$

The relative arrival rate e_{ir} of jobs in the *r*th class at the *i*th node can be calculated directly from routing probabilities as follows:

$$e_{ir} = \sum_{j=1}^{M} \sum_{s=1}^{R} e_{js} \cdot p_{js,ir}\,, \qquad \text{for } i = 1, \dots, M, \quad r = 1, \dots, R. \tag{4.5}$$

And the function $\beta_i(k_i)$ is given by

$$\beta_i(k_i) = \begin{cases} k_i!\,, & k_i \le m_i\,, \\ m_i! \cdot {m_i}^{k_i - m_i}\,, & k_i \ge m_i\,, \\ 1\,, & m_i = 1\,. \end{cases} \tag{4.6}$$

where m_i is the number of active threads in node *i*.

Based on the generation function defined above, relevant performance measures in our model can be written as follows [6]:

Throughput:

$$\lambda_i = \sum_{r=1}^{R} \lambda_{ir} = \sum_{r=1}^{R} e_{ir} \cdot \frac{\mathrm{G}(\mathbf{N} - 1_r)}{\mathrm{G}(\mathbf{N})}, \qquad \text{for } \; i = 1, \dots, M \tag{4.7}$$

Mean Response Time:

By the little's law,

$$T_i = \frac{l_i}{\lambda_i} \tag{4.8}$$

Where the mean number of jobs at the *i*th queuing server l_i is,

$$l_i = \sum_{r=1}^{R} l_{ir}$$

$$= \sum_{r=1}^{R} \sum_{\sum_{j=1}^{M} \mathbf{S}_j = \mathbf{N} \,\&\, \mathbf{S}_i = \mathbf{k}} k_{ir} \cdot f_i(\mathbf{k}) \cdot \frac{\mathrm{G}^{(i)}(\mathbf{N} - \mathbf{k})}{\mathrm{G}(\mathbf{N})}, \quad \text{for } i = 1, \ldots, M \tag{4.9}$$

Where $\mathrm{G}^{(i)}$ can be interpreted as the generation function with the queuing server *i* removed from the network.

We note that the generation function G and the resulting performance measures are defined in the entire design space in Figure 2.4 (with the first order approximation of the program-dimension or workload space). As a result, a salient feature of our analytical modeling approach is its ability to explore the general performance properties of the design space analytically, just like the analysis of the general properties of functions in a multidimensional space in function analysis. Since understanding the general performance properties over the entire design space results in the understanding of the performance at any point in the design space, this approach is particularly useful for the performance analysis in the initial programming phase, when a desirable design choice/point must be identified from a large number of possible choices/points in the design space. Due to the multiplicative increase of computational complexity of G in terms of both cores and threads, however, this approach becomes computationally too expensive as the numbers of cores and threads per core reach a few dozen. The iteration Algorithm introduced in the following section can overcome this limitation.

4.2 Iteration Algorithm

The difficulty for calculating *G*(*x*) lies in the fact that different cores interact with one another through shared resources. A key intuition is that *the effect on each core due to resource sharing would become more and more dependent on the first order statistics (i.e., mean values) and less sensitive to the higher order statistics (e.g., variances) or the actual distributions, as the number of cores sharing the resources increases* (reminiscent of Law of Large Numbers and Central Limit Theorem in statistics and the Mean Field Theory in physics, although actual

formal analysis could be difficult). With this observation in mind, we were able to design an iterative procedure to decouple the interactions among cores, so that the performance of each core can be evaluated quickly as if it were a stand-alone core.

Assume there are N_C cores sharing M common FIFO memories. Initially, we calculate the sojourn time $T_i(0)$ and throughput $\lambda_{ij}(0)$ for single core system i consisting of a core i and the common memories for $i = 1, \dots, N_C$ and $j = 1, \dots, M$. Then the initial mean sojourn time for all the cores, T^*, is calculated over all the single core systems and the routing probabilities, q_j , for the aggregate core to memory j are measured by arrival rate to the memory relative to total arrival rates as in the following iteration formulae:

$$T^*(n) = \sum_{i=1}^{N_C} \frac{\lambda_i(n)}{\sum_{k=1}^{N_c} \lambda_k(n)} T_i(n), \tag{4.10}$$

$$\text{where } \lambda_i(n) = \sum_{j=1}^{M} \lambda_{ij}(n)$$

$$q_j = \sum_{i=1}^{N_C} \frac{\lambda_{ij}}{\sum_{k=1}^{N_C} \lambda_k(n)} \quad , \quad for\ j = 1, \dots, M \tag{4.11}$$

Where q_j is the probability to access the shared memory j from the aggregate core.

Then we enter an iteration loop as shown in Figure 4.1. At the n-th iteration, first the average sojourn times for the common memories, $T_{mj}(n)$, are calculated based on a queuing network on the left of Figure 4.1, including queuing servers for the common memories and an M/M/∞ queuing server characterized by the mean service time $T^*(n)$. There are a total of $N = \sum_{\mathrm{i}=1}^{N_C} m_i$ threads circulating in this network, where m_i is the number of active threads from core i. In other words, we approximate the aggregate effect of all the threads from all the cores

on the common memories using a single M/M/∞ queuing server with the mean service time $T^*(n)$ and routing probabilities to the common memories from the M/M/∞ queuing server. Then, we test if $|T_{mj}(n) - T_{mj}(n-1)| \leq \varepsilon$ holds for all *j* with a predefined small value ε. If it does, exit the loop and finish, otherwise do the following. The sojourn time $T_i(n)$ and throughput $\lambda_i(n)$ for core *i* (for $i = 1, \ldots, N_C$) are updated based on the closed queuing network on the right of Figure 4.1. This time, the effects of other cores on core *i* is approximated by a single M/M/∞ server with the mean service time $T_{mj}(n)$ (for $j = 1, \ldots, M$). There are m_i threads circulating in this network. After all $T_i(n)$ (for $i = 1, \ldots, N_C$) are calculated, $T^*(n)$ will be updated based on the interaction formulae, eq. 4.10 and eq. 4.11, before going to the next iteration. Note that both steps involve only queuing network models that have closed-form solutions, which make the iterations extremely fast. The iteration procedure is summarized in Figure 4.2.

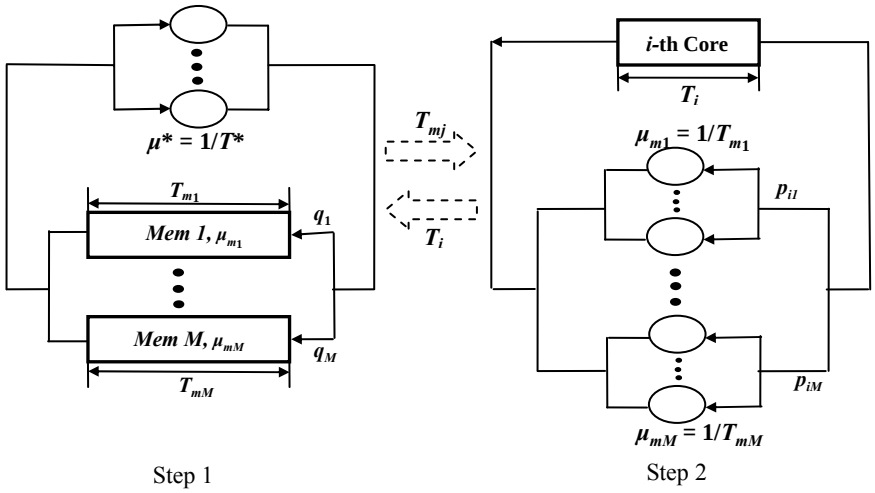

Figure 4.1 An Iterative Procedure for Decoupling Shared Memories

Initialize: ►Get a sojourn time $T_i(0)$ for each and every core in the network consisting of the *i*-th core and the shared memories without considering other cores.

►Calculate the average sojourn time, $T^*(0)$ for the aggregate core using eq. 4.10 and get the routing probabilities to the common memories, $q_j(0)$ using eq. 4.11.

► Set n=1.

Step1: ►Calculate the average sojourn times for the memories, $T_{mj}(n)$ with T^* from previous iteration as in Figure 4.1.

►If $\left|T_{mj}(\mathrm{n}) - T_{mj}(n-1)\right| \leq \varepsilon$ for all j & $n > 2$
Stop Iteration.
Else
Go to Step2.

Step2: ►Get a sojourn time $T_i(n)$ for each core with the $T_{mj}(n)$ from step1 and calculate the $T^*(n)$ and $q_j(n)$.

►Set n = n+1.
Go to Step1.

Figure 4.2 Iteration Algorithm

4.3 Testing

We test the accuracy of the queuing modeling technique by comparing against those obtained by the simulation tool introduced in Chapter 3 in Section 4.3.1 and also explore system behaviors at various points in the design space in Section 4.3.2.

4.3.1 Accuracy Test

Note that the simulation tool must be able to perform a significantly large number of simulation runs (e.g., $10^4 - 10^6$) in a reasonable amount of time for many-core systems (e.g., with 1000 cores). The existing simulation tools are not up to the task. Fortunately, our simulation tool [3] is well suited for the purpose as explained in Chapter 3.

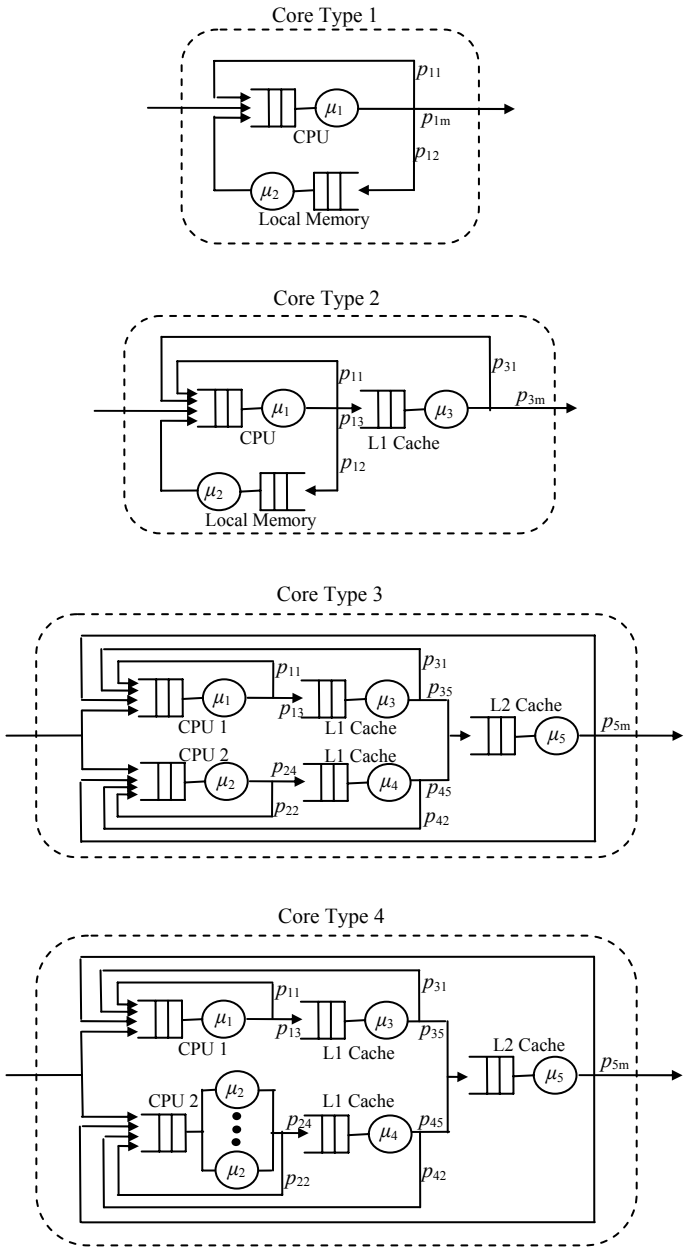

Figure 4.3 Core Types for Testing

To test the performance of the core decoupling procedure, we consider a MP with 1000 cores and core clusters sharing a common FIFO memory. There are two types of cores (Core Type 1 with 6 active threads and Core Type 2 with 9 active threads) and core clusters (Core

Type 3 with 8 active threads and Core Type 4 with 10 active threads), with 250 each, as given in Figure 4.3. Core Type 1 may model a CPU with a local memory (with routing probability p_{12}), routing probability p_{1m} to access the common memory and routing probability p_{11} for events triggering thread switch other than memory accesses. Core Type 2 involves an additional server, modeling, e.g., an L1 cache. Core Type 3 models a two-core cluster with dedicated L1 caches and local shared L2 cache. Group 4 differs from group 3 just in one of its cores, which runs SMT CPU instead of a coarse-grained one (i.e., an *M/M/m* queue and all the rest are *M/M/1* queues).

The parameter settings for each type of cores are given as follows:

- A Common Memory Type = M/M/1

  Service Rate = μ_m
- Core Type 1−group1 (single job (or thread) class):

  $(\mu_1, \mu_2) = (0.05, 0.03)$

  $(p_{11}, p_{12}, p_{1m}) = (0.25, 0.7, 0.05)$

  $m_1=6$
- Core Type 2−group 2 (single job class)

  $(\mu_1, \mu_2, \mu_3) = (0.2, 0.1, 0.1)$

  $(p_{11}, p_{12}, p_{13}, p_{31}, p_{3m}) = (0.3, 0.6, 0.1, 0.95, 0.05)$

  $m_1=9$
- Core Type 3−group 3 (two job classes)

  $(\mu_1, \mu_2, \mu_3, \mu_4, \mu_5)= (0.2, 0.2, 0.2, 0.2, 0.1)$

  $(p_{11}, p_{13}, p_{22}, p_{23}, p_{31}, p_{35}, p_{42}, p_{45}, p_{5m}) =$

  $(0.2, 0.8, 0.2, 0.8, 0.9, 0.1, 0.9, 0.1, 0.02)$

  $(m_1, m_2) = (8, 8)$
- Core Type 3−group4 (two job classes)

  $(\mu_1, \mu_2, \mu_3, \mu_4, \mu_5)= (0.2, 0.1, 0.15, 0.15, 0.1)$

$(p_{11}, p_{13}, p_{22}, p_{23}, p_{31}, p_{35}, p_{42}, p_{45}, p_{5m})$ =

(0.2, 0.8, 0.2, 0.8, 0.9, 0.1, 0.9, 0.1, 0.02)

$(m_1, m_2) = (10, 10)$

For the simulation tool, simulation stops when all nodes in all cores execute at least 10^6 events. For the iteration procedure, ε in the iteration stop condition is set to 0.1 % of $T_m(n)$.

The throughputs for the proposed Queuing Model (QM) and Simulation Tool (ST) at three different common memory service rates are listed in table 4.1 through table 4.3. Both QM and ST are done on an Intel Pentium 4 computer. The computation and simulation times are also listed in the tables.

In each table, there are three columns with different common memory service rates representing three distinctive cases. In the first case, the memory capacity is larger than the aggregate capacity of all cores (i.e., the memory is in under loaded condition). In the second case, their capacities are almost the same. In the last case, the aggregate capacity of all cores is larger than the memory capacity (i.e., the shared memory is potential bottleneck resource). Each column has three sub columns, with the first sub column showing the results from QM and the second from ST and the last the difference between the two. Table 4.1 gives the results for the above parameter settings. Table 4.2 and 4.3 give the results with some parameter changes (see the parameter changes at the top of each table).

Table 4.1 All and Each Type's Throughputs with Various Common Memory Service Time

μ_m	**1.0**			**0.95**			**0.90**		
	QM	***ST***	***Error***	***QM***	***ST***	***Error***	***QM***	***ST***	***Error***
λ_m	9.79E-1	9.64E-1	1.6%	9.50E-1	9.46E-1	0.39%	9.00E-1	9.02E-1	0.24%
λ_{group1}	1.93E-3	1.87E-3	3.2%	1.81E-3	1.80E-3	0.60%	1.62E-3	1.65E-3	1.8%
λ_{group2}	7.98E-4	7.91E-4	0.86%	7.92E-4	7.83E-4	1.1%	7.84E-4	7.81E-4	0.43%
λ_{group3}	6.13E-4	6.10E-4	0.54%	6.13E-4	6.09E-4	0.79%	6.13E-4	6.07E-4	1.1%
λ_{group4}	5.80E-4	5.79E-4	0.18%	5.80E-4	5.78E-4	0.30%	5.80E-4	5.78E-4	0.41%
Run-time	0.2 s 4 loops	3h47m 40s		0.5 s 9 loops	3h46m 6s		0.4 s 8 loops	3h47m 23s	

Table 4.2 Changes in All Cores in Group 1

(μ_1 = 0.05 −> 0.03, m_1 = 6 −> 8)

μ_m	**0.90**			**0.85**			**0.80**		
	QM	***ST***	***Error***	***QM***	***ST***	***Error***	***QM***	***ST***	***Error***
λ_m	8.65E-1	8.59E-1	0.68%	8.50E-1	8.44E-1	0.66%	8.00E-1	8.04E-1	0.45%
λ_{group1}	1.47E-3	1.45E-3	0.99%	1.42E-3	1.42E-3	0.15%	1.25E-3	1.27E-3	1.8%
λ_{group2}	7.98E-4	7.93E-4	0.64%	7.87E-4	7.85E-4	0.16%	7.60E-4	7.60E-4	0.16%
λ_{group3}	6.13E-4	6.09E-4	0.80%	6.13E-4	6.09E-4	0.77%	6.13E-4	6.03E-4	1.7%
λ_{group4}	5.80E-4	5.80E-4	0.02%	5.80E-4	5.79E-4	0.26%	5.80E-4	5.78E-4	0.40%
Run-time	0.3s 6 loops	3h46m 55s		1.3s 26 loops	3h47m 34s		0.6s 12 loops	3h45m 10s	

Table 4.3 A Change in One Target Core in Group 4

(L3 cache hit rate 98% −> 90%)

μ_m	**1.0**			**0.85**			**0.80**		
	QM	***ST***	***Error***	***QM***	***ST***	***Error***	***QM***	***ST***	***Error***
λ_m	8.67E-1	8.62E-1	0.39%	8.50E-1	8.47E-1	0.34%	8.00E-1	8.05E-1	0.62%
λ_{group1}	1.47E-3	1.46E-3	0.82%	1.42E-3	1.42E-3	0.12%	1.24E-3	1.28E-3	4.1%
λ_{group2}	7.99E-4	7.94E-4	0.91%	7.88E-4	7.86E-4	0.24%	7.58E-4	7.57E-4	0.23%
λ_{group3}	6.13E-4	6.11E-4	0.45%	6.13E-4	6.09E-4	0.69%	6.13E-4	6.01E-4	2.3%
λ_{group4}	5.80E-4	5.78E-4	0.36%	5.80E-4	5.79E-4	0.17%	5.80E-4	5.76E-4	0.63%
$\lambda_{group4\text{-}target}$	2.90E-3	2.89E-3	0.42%	2.90E-3	2.89E-3	0.28%	2.90E-3	2.83E-3	2.6%
Run-time	0.2s 3 loops	3h46m 53s		0.5s 10 loops	3h47m 11s		0.6s 12 loops	3h48m 3s	

It turns out that our iterative procedure is highly accurate and fast. For the cases in the tables 4.1 and 4.3, it takes less than 12 iterations to get the results. For the case in table 4.2, the number of iterations increases up to 26. For all the cases studied, the technique is three orders of magnitude faster than ST, finishing within a few seconds. This allows a large design

space (or parameter space) to be scanned numerically. Moreover for all the cases, the results are consistently within 5 % of the simulation results.

One can further reduce the time complexity by running the iterative procedure only once for a given set of parameters to get effective T_m. Then study the performance for any target core in a range of workload parameters based on the closed queuing network in Figure 4.1(the one on the right) with T_m fixed. One can expect that changing parameters for just one core out of 1000 cores should not significantly affect T_m as is evidenced from the case study in table 4.3 compared with table 4.2. This approximation can further reduce the time complexity by another two orders of magnitude.

As one can see, with the proposed technique, the closed queuing network models can now be effectively used to explore the design space in Figure 2.4. A user of our technique may start with a coarse-grained scanning of the entire space first, which will allow the user to identify areas of further interests in the space. Then perform a finer grained scanning of the areas of interests.

4.3.2 Example of Design Space Exploration

In this section, we demonstrate how the proposed solution can be used for comparative study of different processor architectures in a large workload space.

We consider three different types of processors as given in Figure 4.4, including (a) a multithreading (MT) processor; (b) a simultaneous multithreading (SMT) processor; and (c) a multicore processor (MP). The MT processor in Figure 4.4 (a) consists of a coarse-grained CPU, a local memory with routing probability p_{12} and a cache with routing probability p_{13}. The SMT processor in Figure 4.4 (b) is the same as the MT processor except that it runs a SMT CPU, rather than a coarse-grained one. The MP processor in Figure 4.4 (c) runs three MT cores in a die in parallel. All three types of processors have a single off-chip memory.

Obviously, we can expect that with the same CPU powers, both SMT and MP processors will outperform the MT processor performance. Hence, the MT processor provides a

baseline reference against which the performance gain of the other two types of processors can be measured. Given the complex interactions among threads and cores due to the sharing of on-chip caches and an off-chip memory and a large workload space, how effective SMT and MP are able to enhance the system performance over the baseline is a challenging question to answer. Our modeling technique is particularly viable to tackle such questions.

In our study, we consider two distinct workload types, i.e., CPU-bound (CB) and memory-bound (MB). For CB, $\mu_m = \mu_1$, and for MB, 10 $\mu_m = \mu_1$. In other words, for MB, the mean service time at the off-chip memory is ten times longer than that for CB, meaning that the latter one has ten times of workload on the memory than the former one.

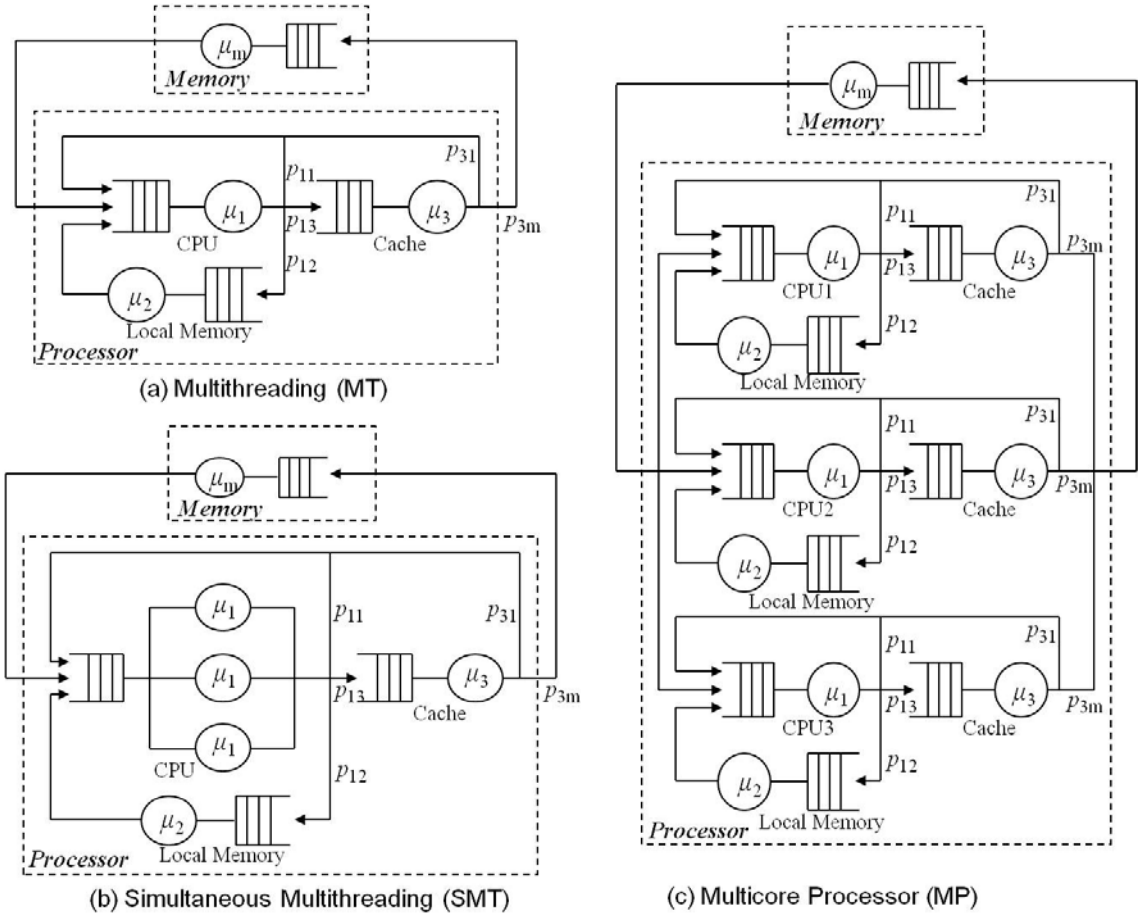

Figure 4.4 Processor Types

In summary, we explore 12 points in the design space: 2 (CB or MB)*2 (with cache or without) * 3 (MT, SMT, or MP).

The parameter settings for each type of processors are given as follows:

- MT :

  $(\mu_1, \mu_2, \mu_3) = (1, 3, 10)$, $\mu_m = 1$ (*CB*)

  $(\mu_1, \mu_2, \mu_3) = (1, 3, 10)$, $\mu_m = 0.1$ (*MB*)

- SMT:

  $(\mu_1, \mu_2, \mu_3) = (1, 3, 10)$, $\mu_m = 1$ (*CB*)

  $(\mu_1, \mu_2, \mu_3) = (1, 3, 10)$, $\mu_m = 0.1$ (*MB*)

- MP:

  3MT processors in parallel, $\mu_m = 1$ (*CB*)

  $\mu_m = 0.1$ (*MB*)

- Routing probabilities: $(p_{11}, p_{12}, p_{13}, p_{31}, p_{3m}) = (0.1, 0.5, 0.4, 0.8, 0.2)$

  Without Cache, routing probabilities will be:

  $(p_{11}, p_{12}, p_{1m}) = (0.1, 0.5, 0.4)$

  The workloads on all threads have the same statistic characteristics (same code path)

All service rates are relative to the CPU service rate (=1). The off-chip and CPUs in MT and MP are modeled as M/M/1 queue and the CPU in SMT is modeled as M/M/m queue. The processor models are solved analytically.

First, we study the overall system throughput performance as a function of the number of threads for all 12 points. The numerical results are plotted in Fig. 4.5. There are four subplots corresponding to the four points in the workload and cache dimensions.

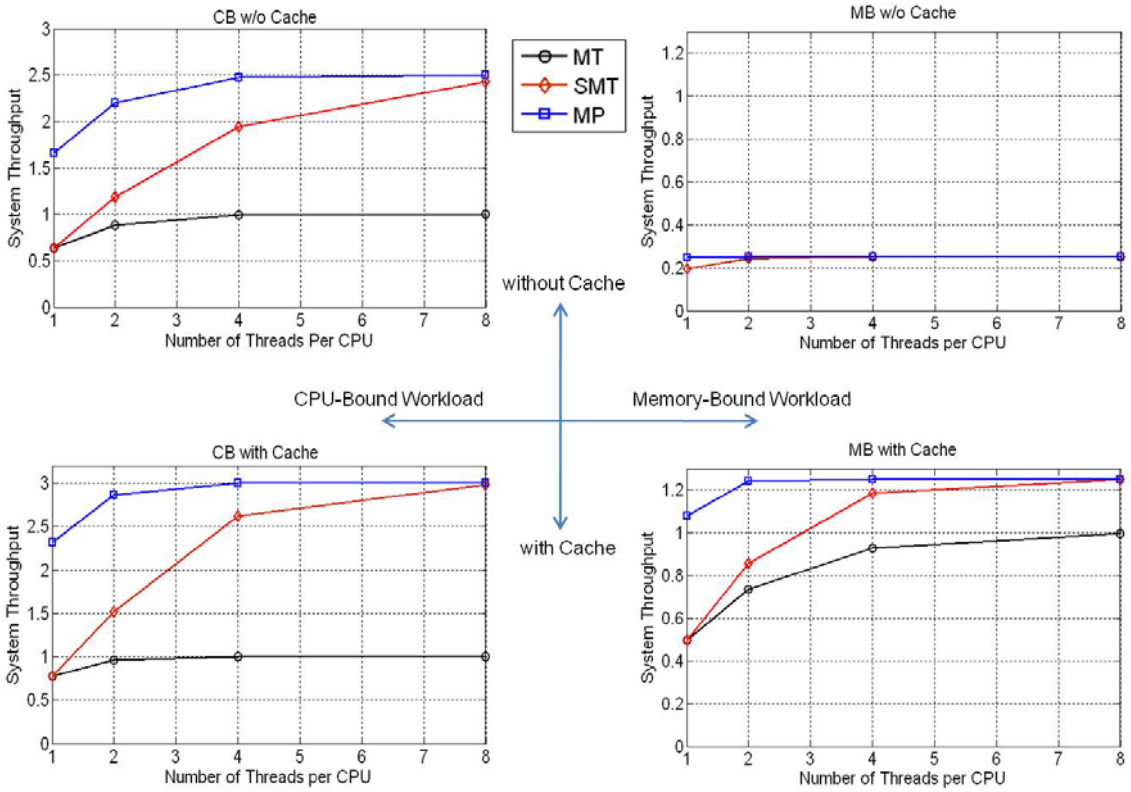

Figure 4.5 Throughputs in MT, SMT, and MP with Multidimensional Changes

First, we note that the highest possible throughputs for MT, SMT, and MP are 1, 3, and 3, respectively. They are reached for the case in the lower left plot in Figure 4.5, when the workload is CB; the cache exists; and the number of threads reaches 8. Moreover, when the number of threads is less than 8, MP outperforms SMT, which is the case for all other scenarios as well, at however a much higher cost (three cores versus a single core with three issues per clock cycle). This means that for the CB workload, cache and multithreading can effectively hide the memory access latency from the CPUs, which are particularly useful for improving the SMT processor performance while avoiding higher cost for a multicore solution.

Second, we look at the other extreme, i.e., the workload is MB and the cache does not exist, as is the case in the upper right plot in Figure 4.5. In this case, both SMT and MP are ineffective. The multithreading cannot effectively hide memory latency from the CPUs. Both multiple issues and multicore do not do better than the baseline solution, i.e., MT, which

however, also performs poorly, reaching only 0.22 throughput performance. In this case, adding caching can greatly enhance the baseline as well as SMT and MP, as is evident in the lower right plot in Figure 4.5.

Finally, we note that for the CB workload without caching, memory access latency cannot be completely hidden by multithreading, resulting in reduced throughput performance for SMT and MP, as shown in the upper left plot in Figure 4.5. Also in all the cases studied, multithreading helps to only a certain extent. In other words, having more than eight threads cannot help further improve the throughput performance for all the cases.

In summary, a system with multithreading and SMT can be good solutions to achieve high performance at a lower cost than a multicore solution with the number of cores no greater than the number of simultaneous issues for a SMT CPU. Moreover, for SMT and MP to be effective under MB workload, shared resources should not pose bottlenecks that might throttle the overall throughput performance. In addition to multithreading and caching, other mechanisms such as parallel memory accesses using, e.g., memory banks, or pipelined memory access, may need to be used to ensure reasonably high throughput.

As one can see, a user may use our technique to performance a coarse-grained scanning of the entire design space to identify design points of interest. Then performing a finer grained scanning using other techniques to further pin down the optimal design points.

4.4 Related Work

Traditionally, simulation and benchmark testing are the dominant approaches to evaluate the processor performance. Unfortunately, these approaches quickly become ineffective as the number of cores increases. Hence, there have been many alternative approaches in an attempt to address this scalability issue. Statistical simulation (e.g., [28] [29] [30]) makes the short synthetic trace from a long real program trace and save time by simulating the short statistic trace. Partial simulation (e.g., [31] [32] [33]) reduces total simulation time by

selectively measuring a subset of benchmarks. The design space exploration based on intelligent predictive algorithms trained by sampled benchmarks (e.g., [34] [35] [36] [37]) can predict the performance in the entire design space from simulations of a given benchmark from a small set of the design space. However, most existing approaches have focused on the exploration of microarchitectural design space and again quickly become ineffective as the numbers of cores and threads in the system increase. Moreover, the pace at which the multicore architectures proliferate makes it difficult for the existing approaches to keep up, especially in terms of comparative performance analysis of different architectures. Our approach makes it possible to quickly identify the areas of interests in a large design space at coarse granularity, in which the existing finer granularity tool can work efficiently to pinpoint the optimal operation points.

In terms of queuing network modeling, since Jackson's seminal work [38] in 1963 on queuing networks of product form, a wealth of results on the extension of his work has been obtained for both closed and open queuing networks. Notable results include the extensions from M/M/1 FCFS (First-Come-First-Served) to LCFS (Last-Come-First-Served) preemptive resume, PS (Processor Sharing), and IS (Infinite Server) queuing disciplines, multiple job classes (or chains) and class migrations, load-dependent routing and service times, and exact solution techniques such as convolution and Mean Value Analysis (MVA), and approximate solution techniques for queuing networks with or without product form. Sophisticated queuing network modeling tools were also developed, making queuing modeling and analysis much easier. These results are well documented in standard textbooks, tutorials, and research papers (e.g., [4] [6] [11] [39]). As a result, in the past few decades, queuing networks were widely adopted in modeling computer systems and networks (e.g., [39] [40] [41] [42] [43] [44] [45]).

However, very few analytical results are available for multicore processor analysis. The work in [46] introduced a fast analytical modeling technique for multiprocessors. However, it didn't model the multithreading effect, which is at the core of our framework. The work in [7]

modeled the heterogeneous multicore processors with multithreading. But as the number of multithreaded cores increases, the computational complexity quickly becomes unmanageable as mentioned in the paper. The work in [47] was able to accurate model a multithreaded processor using Markovian models. But the model can only be applied to fine-grained CPUs and hence cannot cover a large design space. In [48], a mean value analysis of a multithreaded multicore processor is performed. The performance results reveal that there is a performance valley to be avoided as the number of threads increases, a phenomenon also found earlier in multiprocessor systems studied based on queuing network models [49][50]. Markovian Models are employed in [47] to model a cache memory subsystem with multithreading. However, to the best of our knowledge, the only work that attempts to model multithreaded multicore using queuing network model is given in [51]. But since only one job class (or chain) is used, the threads belonging to different cores cannot be explicitly identified and separated in the model and hence multicore effects are not fully accounted for.

Most relevant to our work is the work in [43]. In this work, a multiprocessor system with distributed shared memory is modeled using a closed queuing network model. Each computing subsystem is modeled as composed of three M/M/1 servers and a finite number of jobs of a given class. The three servers represent a multithreaded CPU with coarse-grained thread scheduling discipline, a FCFS memory, and a FCFS entry point to a cross-bar network connecting to other computing subsystems. The jobs belonging to the same class or subsystem represent the threads in that subsystem. The jobs of a given class have given probabilities to access local and remote memory resources. This closed queuing network model has product-form solution.

The above existing application of queuing results to the multithreaded multicore and multiprocessor systems are preliminary (i.e., within the small cone on the left in Figure 2.4). The only queuing discipline studied is the FCFS queue, which characterizes the coarse-grained thread scheduling discipline at a CPU and FCFS queuing discipline for memory or

interconnection network. No framework has ever been proposed that can cover a design space anywhere near the size as the one in Figure 2.4 and that allow system classes to be analyzed over the entire space.

Moreover, no existing queuing network model is capable of characterizing the dynamics of a program in terms of thread-level parallelism (i.e., different code segments use different numbers of threads). Although the traditional Fork-Join approach [6] can capture such effect, it cannot be applied to multicore systems simply because it assumes that different parallel code branches are handled by different processors, rather than different threads belonging to the same processor or core. Another approach being proposed [6], which is amenable to queuing analysis, is to use a hybrid open-and closed queuing network with two job classes. The job class running in the closed loop emulates a fixed level of parallelism, whereas the job class in the open loop models the dynamics of thread-level parallelism. The problem with this approach is that it is difficult to match the model with the statistics of parallelism of the actual code.

CHAPTER 5

CONCLUSION

This dissertation defined, developed, and tested a theoretical framework for design space exploration of multicore systems. In addition, a fast and accurate thread-level simulation tool was developed, that allows many-core processor performance to be tested quickly for given points in a large design space. This has made the testing of our theoretical framework for many-core processors possible. The novelty of the framework lies in the fact that it works at the thread level and it studies the general properties of system classes over the entire space. Hence, it is free of scalability issues. One additional feature of our framework is its ability to be applied to any system, including real-time, embedded, multicore, manycore, and communication systems.

This dissertation is expected to provide much needed techniques and tools to help better understand MPs and effectively aid the analysis, design, and programming of MPs. Our approach is expected to strengthen the awareness in the research community and industry of the importance of analytical approaches in guiding the design of robust complex systems. The queuing modeling techniques developed in the dissertation will make a contribution to the advancement of applications of queuing theory.

APPENDIX A

SAMPLE CODE PATHS

Table A.1 Code Path for Packet Count (IXP 1200)

Task	# instructions in code segment ($t_{m,k}$-$t_{m-1,k}$)	Type of I/O Access	Unloaded latency $\tau_{j,k}$
Poll to get control of receive state machine	11	Voluntary yield	0
Check receive ready flag	4	FBI CSR read	15
Issue receive request and move packet from MAC to RFIFO	11	FBI CSR write & IX bus read	84
Check receive control register	1	FBI CSR read	14
Check for proper reception & Allocate buffer	9	SRAM read	21
Move packet to buffer	25	SDRAM read	50
Increment packet count	9	Scratch read	14
Increment byte count	3	Scratch write	17
Release buffer handle	2		
total	75		215

Table A.2 Code Path for ATM/Ethernet IP Forwarding (IXP 1200)

Task	# instructions in code segment ($t_{m,k}$-$t_{m-1,k}$)	Type of I/O Access	Unloaded latency $\tau_{j,k}$
Get port ready (CSR read)	7	FBI read	12
Start receiving packet from MAC to RFIFO	5	FBI write & IX Bus receive	92
Move packet from IX Bus to RFIFO - Switch context for completion of packet reception	4	FBI write & IXBus receive	76
Read Control Register	2	FBI read	12
Get buffer descriptor from SRAM (memory address to store packet) - Switch context for completion of SRAM access	11		0
If a buffer is available read the packet from RFIFO to DRAM	17	RFIFO read	15
Complete reading to buffer from RFIFO	14	RFIFO read	20
Verify TTL, Check sum	18	SRAM read	15
IP route lookup	27		0
Get TRIE pointer	7	SRAM read	15
Get TRIE pointer	5	SRAM read	15
Get TRIE pointer	5	SRAM read	15
Get TRIE pointer	5	SRAM read	15

Wait for SDRAM access completion - Switch context for completion of SDRAM access	4		0
Forward to an ATM port	1	SDRAM write	46
Forward to an ATM port	15	SRAM write	17
Enqueue ATM AAL5 PDU (calculate # of bytes to pad, write AAL5 trailer) - Switch context for completion of SDRAM access	34	SDRAM read	47
Write PDU - Switch context for completion of SDRAM access	9	SRAM write	14
Enqueue ATM – begin	12	SRAM read	19
Enqueue ATM – complete - Switch context for completion of SRAM access	20	SRAM write	20
Miscellaneous	6		
Total	228		465

Table A.3 Code Path for Layer-2 Filtering (IXP 1200)

Task	# instructions in code segment ($t_{m,k}$–$t_{m-1,k}$)	Type of I/O Access	Unloaded latency $\tau_{j,k}$
Check CSR for data in the port	4	FBI read	15
Write control word to CSR and start receiving packets	6	FBI write &FBI read	84
Read recive control information (to check status of packet receive operation from port)	1	FBI read	15
Swap context and wait to receive buffer address for packet from SRAM	11		
Read from SRAM (3 consecutive memory locations)	20	SRAM read	23
Read from RFIFO	6	RFIFO read	26
Hash source address and destination address	22	RFIFO read	37
SRAM lookup (Source Address (SA), Destination Address (DA) and hash value)	9	ScratchPad write	19
Read the forwarding table for DA in SDRAM (4 long words)	14	SRAM read	47
Check for Bridging and isolate DA from forwarding table entry	9	SDRAM read	25
Read the forwarding table for DA in SDRAM (4 long words)	20	SRAM read	57
Do Layer 2 packet filtering (Packet filtering rules obtained from forwarding table in SRAM)	43	SDRAM read	48
Check for EOP and Packet discard bit information in packet receive state	10	SRAM write	21
Enqueue packet (2 long words)	5	SRAM read	23
If empty queue create a queue for enqueuing packet	14	SRAM write	22

Miscellaneous	5		
total	**199**		**462**

Table A.4 Code Path for *Generic IPv4 Forwarding* (IXP 1200)

Task	# instructions in code segment ($t_{m,k} - t_{m-1,k}$)	I/O	$\tau_{j,k}$
Check receive ready flags	5	FBI read	14
Move packet from IX Bus to RFIFO	8	FBI write & IX Bus receive	76
Read receive control information	2	FBI read	19
Wait for buffer allocation in SDRAM; get the descriptor from SRAM	11	SRAM read	17
Read 3 Quad words from RFIFO into ME for IP validation	16	RFIFO read	18
Read 2nd 32 byte to SDRAM(in the allocated buffer)	15	RFIFO read	22
IP lookup	40	SRAM read	17
IP lookup	7	SRAM read	17
IP lookup	5	SRAM read	17
Go next hop information from SDRAM	7	SDRAM read	47
Write packet descriptor to SRAM (after associating it with a TX port)	16	SRAM write	18
Read queue descriptor from SRAM (for enqueue operation)	4	SRAM read	22
Write the packet descriptor to SRAM(to the TX queues associated with the TX port)	15	SRAM write	20
Miscellaneous	6		
total	157		324

Table A.5 Code Path for *Diffserv POS* (IXP2400)

Task	# instructions in code segment ($t_{m,k} - t_{m-1,k}$)	I/O	$\tau_{j,k}$
PPP decapsulation, PPP classifier and DSCP classifier	28	SRAM read	88
6-tuple classifier	35	Hash operation	102
Get hash table entry using a hash key as an index	4	SRAM read	93
Free the hash entry and check if the IP is valid	35	SRAM Test and add	103
Update test_byte_count	3	yield	0
DSCP Marker and IPv4 forwarder	110	SRAM read	97

Allocate transfer registers and store packet into them	60	SRAM & DRAM write	48
Send the packet to the next block through scratch ring	22	Scratch put & get	87
Read the packet header and cache it in local memory	15	DRAM read	148
Total	312		766

Table A.6 Code Path for *MPLS* (IXP2400)

Task	# instructions in code segment ($t_{m,k} - t_{m-1,k}$)	I/O	$\tau_{j,k}$
Reads packet header from SDRAM and caches aligned header to local memory	17	DRAM read	156
PPP decapsulation, PPP classifier and DSCP classifier	97	SRAM read	84
First Lookup on destination	20	3SRAM reads	83
Second Lookup	11	SRAM read	74
Third Lookup	9	SRAM read	76
Read and process the Next hop information	10	SRAM read	78
Process MPLS packet, forward ipv4 and write back IP header to DRAM and write back meta-data to SRAM	126	DRAM &SRAM write	38
Put ring and sink data to next block and get the data from the previous block for packet availability for MPLS processing	17	Scratch put & get	77
Total	311		666

Table A.7 Code Path for IPv4 Ethernet (IXP2400)

Task	# instructions in code segment ($t_{m,k} - t_{m-1,k}$)	Type of I/O access	Unloaded Latency $\tau_{j,k}$
Reads packet header from DRAM and caches aligned header to local memory	16	DRAM read	116
Ethernet frame destination MAC filtering, decap and classification. IPv4 Forwarder: IPv4 total length verification, IPv4 Checksum verification, Ipv4 Address verification, Extract source address. Fetch directed broadcast table entry, which is in the SRAM	100	SRAM read	100
Perform lookup on destination address. IPv4 Trie5 lookup. It uses maximum 5 SRAM references. First Lookup	20	SARM read	84
Second Lookup	11	SRAM read	78

Third Lookup	9	SRAM read	81
Fourth Lookup	9	SRAM read	79
Fifth Lookup	9	SRAM read	79
Read and process the Next hop information	10	SRAM read	84
Sets the next hop id, Set the fabric port. This more like a blade ID, IPv4 update header (Decrement TTL, update checksum and place the packet in destination buffer). Write back IP header to DRAM. Write back meta-data to SRAM.	90	DRAM write & SRAM write	45
Check ring and sink data with (buffer handle, last buffer in the chain and queue number) for next block(Queue manager).Check the ring from the previous block (Packet Receive stage) for packet availability for Ethernet/IP processing.	21	SCRATCH put & get	84
Total	295		830

REFERENCES

[1] E. Johnson and A. Kunze, “IXP 1200 Programming,”, Intel Press.

[2] Cavium Octeon CN38xxx (http://www.cavium.com/) and SUN UltraSPARC T2 (http://www.sun.com).

[3] H. Jung, M. Ju, H. Che, and Z. Wang, “A Fast Performance Analysis Tool for Multicore, Multithreaded Communication Processors,” in Proc. of the 11th IEEE High Assurance Systems Engineering Symposium (HASE), Dec. 2008.

[4] “Teletraffic Engineering,” ITU-D Study Group 2, 2002.

[5] D. Towsley, “Queuing Network models with state-dependent routing,” Journal of ACM, Vol. 27, No. 2, pp. 323-337, Apr. 1980.

[6] G. Bolch, S. Greiner, H. de Meer and K. S. Trivedi, “Queueing Networks and Markov Chains,” John Wiley, 2nd edition, 2006.

[7] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, “Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance,” in Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA’04), 2004.

[8] Intel, IXP1200 developer workbench, provided by IXA SDK 2.x

[9] Intel, IXP2400 developer workbench, provided by IXA SDK 3.x

[10] H. Che, C. Kumar, and B. Menasinahal, “Fundamental Network Processor Performance Bounds,” the 4th IEEE International Symposium on Network Computing and Applications (IEEE NCA05), 2005.

[11] I. F. Akyildiz and A. Sieber, “Approximate Analysis of Load Dependent General Queuing Networks,” IEEE Transactions on Software Engineering, Vol. 14, No. 11, pp. 1537-1545, Nov. 1988.

[12] T. Austin, E. Larson, and D. Ernst, “SimpleScalar: an Infrastructure for Computer System Modeling, “ IEEE Computer, Vol. 35, No. 2, 2002.

[13] M. Rosenblum, E. Bugnion, S. Devine, S. A. Herrod, “Using the SimOS Machine Simulator to Study Complex Computer Systems,” Modeling and Computer Simulations, Vol. 7, No. 1, pp. 78-103, 1997.

[14] E. Kohler, R. Morris, B. Chen, J. Janotti, and M. F. Kaashoek, “The Click Modular Router,” ACM Transactions on Computer Systems, Vol. 18, No. 3, pp. 263-297, Aug. 2000.

[15] Z. Huang, J. P. M. Voeten, and B. D. Theelen, “Modeling and Simulation of a Packet Switch System using POOSL,” Proceedings of the PROGRESS workshop 2002, pp. 83-91, Oct. 2002.

[16] W. Xu and L. Peterson, “Support for Software Performance Tuning on Network Processors”, IEEE Network, July/Aug. 2003.

[17] Y. Luo, J. Yang, L. N. Bhuyan, and L. Zhao, “NePSim: A Network Processor Simulator with a Power Evaluation Framework,” IEEE Micro, Special Issue on Network Processors for Future High-End Systems and Applications, Vol. 24, No. 5, pp. 34 – 44, Sept/Oct. 2004.

[18] P. Paulin, C. Pilkington, and E. Bensoudane, “StepNP: A System-Level Exploration Platform for Network Processors,” IEEE Design & Test of Computers, Vol. 19, No. 6, pp. 17-26, Dec 2002.

[19] “Advanced Software Development Tools for Intel IXP2xxx Network Processors,” Intel White Paper, Oct. 2003.

[20] L. Thiele, S. Chakraborty, M. Gries, S. Kiinzli, “Design Space Exploration of Network Processor Architectures,” Network Processor Design: Issues and Practices, Editors: P.

Crowley, M. Franklin, H. Hadimioglu, P. Onufryk, Morgan Kaufmann Publishers, October 2002.

[21] J. Yao, Y. Luo, L. Bhuyan, and R. Iyer, “Optimal Network Processor Topologies for Efficient Packet Processing,” in Proc. of IEEE GLOBECOM, Nov. 2005.

[22] R. Ramaswamy, N. Weng, and T. Wolf, “Analysis of Network Processing Workloads,” in Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 226-235, Austin, TX, March 2005.

[23] N. Weng and T. Wolf, “Pipelining vs Multiprocessors – Choosing the Right Network Processor System Topology,” in Proc of Advanced Networking and Communications Hardware Workshop (ANCHOR 2004) in conjunction with ISCA 2004, June 2004.

[24] L. Yang, T. Gohad, P. Ghosh, D. Sinha, A. Sen, and A. Richa, “Resource Mapping and Scheduling for Heterogeneous Network Processor Systems,” in Proc. of Symposium on Architecture for Networking and Communications Systems, 2005.

[25] J. Emer, et. al., “Asim: A Performance Model Framework,” Computer Magazine, pp. 67, Feb. 2002.

[26] M. Vachharajani, N. Vachharajani, D. Penry, J. Blome, and D. August, “The Liberty Simulation Environment,” Version 1.0.

[27] F. Baskett, K.M. Chandy, R.R. Muntz and F. Palacios, “Open, Closed, and Mixed Networks of Queues with Different Classes of Customers”, Journal of the ACM 22, No. 2, pp. 248–260, Apr. 1975.

[28] D. Genbrugge and L. Eeckhout, “Chip Multiprocessor Design Space Exploration through Statistical Simulation”, IEEE Transactions on Computers, Vol. 58, No. 12, pp. 1668-1681, Dec. 2009

[29] C. Hughes and T. Li, “Accelerating Multi-Core Processor Design Space Evaluation Using Automatic Multi-Threaded Workload Synthesis,” Proc. IEEE Int’l Symp. Workload Characterization (IISWC), pp. 163-172, Sept. 2008.

[30] S. Nussbaum and J.E. Smith, “Statistical Simulation of Symmetric Multiprocessor Systems,” Proc. 35th Ann. Simulation Symp., pp. 89-97, Apr. 2002.

[31] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. “Automatically Characterizing Large Scale Program Behavior.” Intl. Symp. On Architectural Support for Programming Languages and Operating Systems, pp. 45-57, Dec. 2002.

[32] R. Wunderlich, T. F. Wenish, B. Falsafi, and J. C. Hoe. “SMARTS: Accelerating Microarchitectural Simulation via Rigorous Statistical Sampling” Intl. Symp. on Computer Architecture, pp. 84-95, June 2003.

[33] T. Wenisch, R. Wunderlich, B. Falsafi, and J. Hoe, “TurboSMARTS: Accurate microarchitecture simulation sampling in minutes,” SIGMETRICS Performance Evaluation Review,33, 1, pp.408–409, 2005

[34] E. Ipek , S. A. McKee , K. Singh , R. Caruana , B. R. de Supinski, and M. Schulz, “Efficient architectural design space exploration via predictive modeling,” ACM Transactions on Architecture and Code Optimization, Vol.4 No.4, pp.1-34, Jan. 2008

[35] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra, “Using Predictive Modeling for Cross-Program Design Space Exploration in Multicore Systems”, in Proc. of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT), Sept. 2007

[36] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. “A Predictive Model for Superscalar Processor Performance.” Intl. Symp. on Microarchitecture, pp. 161-70, Dec. 2006.

[37] B. C. Lee and D. M. Brooks. “Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction.” Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, pp 185-194, Oct. 2006.

[38] J. R. Jackson, “Jobshop-like Queuing Systems,” Management Science, Vol. 10, pp. 131-142,1963.

[39] A. Thomasian and P. F. Bay, “Integrated Performance Models for Distributed Processing in Computer Communication Networks,” IEEE Transactions on Software Engineering, Vol. SE-11, No.10, pp. 1203-1216, Oct. 1985.

[40] P. P. Chen, “Queuing Network Model of Interactive Computing Systems,” in Proc. of the IEEE, Vol. 63, No. 6, June 1975.

[41] D. Ghosal and L. Bhuyan, “Performance Evaluation of a Dataflow Architecture,” IEEE Transactions on Computers, Vol. 39, No. 5, pp. 615 – 627, May 1990.

[42] N. Lopez-Benitez and K. S. Trivedi, “Multiprocessor Performability Analysis,” IEEE Transactions on Reliability, Vol. 42, No. 4, pp. 579-587, Dec. 1993.

[43] S. S. Nemawarkar, R. Govindarajan, G. R. Gao, and V. K. Agarwal, “Analysis of Multithreaded Multiprocessors with Distributed Shared Memory,” in Proc. of the Fifth IEEE Symposium on Parallel and Distributed Processing, Dec. 1993.

[44] B. Smilauer, “General Model for Memory Interference in Multiprocessors and Mean Value Analysis,” IEEE Transactions on Computer, Vol. 34, No. 8, pp. 744-751, Aug. 1985.

[45] M. Woodbury and K. G. Shin, “Performance Modeling and Measurement of Real-Time Multiprocessors with Time-Shared Buses,” *IEEE Transactions on Computers*, Vol. 37, No. 2, pp.214-224, Feb. 1988.

[46] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vermon, and D. A. Wood, “Analytic Evaluation of Shared-memory System with ILP Processors,” in Proceedings of the 25th Annual InternationalSymposium on Computer Architecture (ISCA’98), 1998.

[47] X. E. Chen and T. M. Aamodt, “A First-Order Fine-Grained Multithreaded Throughput Model,” in Proc. of the 15th IEEE International Symposium on High-Performance Computer Architecture (HPCA), Feb. 2009.

[48] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, “Many-Core vs. Many-Tthread Machines: Stay Away From the Valley,” IEEE Computer Architecture Letter, vol. 8, no. 1, pp. 25-28, Jan. 2009.

[49] A. Agarwal, “Performance Tradeoffs in Multithreaded Processors,” IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 5, pp. 525-539, Sept. 1992.

[50] R. Saavedra-Barrera, D. Culler, and T. von Eicken, “Analysis of Multithreaded Architectures for Parallel Computing,” in the Proc. of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 169-178, 1990.

[51] V. Bhaskar, “A Closed Queuing Network Model with Multiple Servers for Multithreaded Architecture,” Journal of Computer Communications, Vol. 31, pp. 3078-89, 2008.

[52] H. Jung, M. Ju, and H. Che. “A Theoretical Framework for Design Space Exploration of Manycore Processors,” in the Proc. of the 19th IEEE Annual International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), pp.117-125, July 2011.

BIOGRAPHICAL INFORMATION

Hun Jung is currently a Ph.D. Student in the Department of Computer Science and Engineering at the University of Texas at Arlington. He received his B.S. degree from Korea University in Seoul, South Korea in 2002, his M.S. degree from State University of New York at Buffalo, New York in 2005, and his Ph.D. degree from the University of Texas at Arlington in 2011. His current research interest is in the area of performance design and analysis for Multicore processors including communication processors. He is currently a member of IEEE society.