A CLASS OF QUEUING NETWORK MODELS FOR MULTITHREADED PROCESSORS

by

MIAO JU

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

August 2011

To my parents

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervising professor, Dr. Hao Che, for his patience, encouragement and advice during my research. His strong understanding of the subject and constant motivation made this work possible. I would like to thank him for giving me the opportunity to work in his research group. I would also like to thank Dr. Bob Weems, Dr. Jeff Lei and Dr. Yonghe Liu for being on my committee.

I take this opportunity to thank all my friends at UTA, especially Hun Jung in our lab for his valuable discussions and support. I also would like to thank both Mr. Xing Zhao and Yang Liu from Columbia University for their helpful suggestions.

Finally, I would like to thank my parents for their constant encouragement and support without which this work would have not been possible.

July 15, 2011

ABSTRACT

A CLASS OF QUEUING NETWORK MODELS FOR MULTITHREADED PROCESSORS

MIAO JU, Ph.D.

The University of Texas at Arlington, 2011

Supervising Professor: Hao Che

With ever expanding design space and workload space in multicore era, a key challenge to program a multithreaded multicore processor is how to evaluate the performance of various possible program-task-to-core mapping choices and provide effective resource allocation during the initial programming phase, when the executable program is yet to be developed. In this dissertation, we put forward a thread-level modeling methodology to meet this challenge. The idea is to model thread-level activities only and overlook the instruction-level and microarchitectural details. A model developed at this level assumes the availability of only a piece of pseudo code that contains information about the thread-level activities, rather than an executable program that provides instruction-by-instruction information. Moreover, 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.

The above features make the methodology particularly amenable for fast performance evaluation of a large number of program-task-to-core mapping choices during the

initial programming phase. Based on this methodology, in this dissertation we further developed: 1) an analytic modeling technique based on queuing theory which allows large design space exploration; and 2) a framework that allows program tasks to be mapped to different core resources to achieve maximal throughput performance for many-core processors. Case studies against cycle-accurate simulation demonstrate that the throughput estimated using our modeling technique is consistently within 8% of cycle-accurate simulation results.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

As chip multiprocessors (CMPs) become the mainstream processor technology, challenges arise as to how to partition application tasks and map them to one of many possible core/thread configurations (i.e., program-task-to-core mapping) to achieve desired performance in terms of e.g., throughput, delay, power, and resource consumptions. There are two scalability barriers that the existing CMP 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 CMP 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 CMPs proliferate. The first barrier is particularly problematic for communication processors (CPs) . First, to maximize the throughput performance, a CP generally employs massive core-level pipeline and parallelism for packet processing. The number of possible pipeline/parallel configurations grows exponentially as the number of cores increases. Second, the workload for a CP is a complex function of both packet arrival process and mixture of code paths at the thread level and there are virtually unlimited number of workloads to be tested as a result of this. The existing CP analysis tool cannot help identify what ”typical” workloads should be tested, which are assumed to be determined by the user of the tool, rather than part of the tool design. This makes initial performance analysis of program-task-to-core mapping using an existing tool extremely difficult. In addition to these barriers, how to analyze the performance of various possible design/programming choices during the initial CMP

design/programming phase is particularly challenging, when the actual instruction-level program is not available.

To meet the above challenges, a solution must satisfy the following three stringent requirements. First, it must be fast enough to allow a potentially large number of mapping choices to be tested within a reasonable amount of time. Second, the performance data it provides must be reasonably accurate. Third, it must not assume the availability of executable programs as input for testing. These three requirements pose significant challenges to the development of such a solution. On one hand, to provide reasonably accurate performance data, the solution must take into account processing details that may contribute significantly to the system-level performance, such as the thread-level activities, and memory and I/O resource contentions at the thread, core, and system levels. On the other hand, indiscriminately including various processing details in such a solution can quickly slow down the processing, rendering the solution useless.

In this dissertation, we present a novel CMP analysis methodology to meet the above requirements (Chapter 2 and [26][27]). In the proposed methodology, two unique features are employed to overcome the scalability barriers. First, the methodology is coarse-grained, which works at the thread level, and overlooks instruction-level and microarchitectural details, except those having significant impact on thread level performance. This coarse granularity is particularly amenable to large design space exploration and theoretical analysis. Second, the approach taken for the design space exploration in our methodology is unconventional. Instead of exploring the design space based on sampled points in the space, the methodology directly study the general performance properties of system classes over the entire design space. 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.

Based on the proposed methodology, a simulation tool, a queuing network modeling technique and a framework for fast program-task-to-core mapping are developed. The simulation tool is generic, in the sense that it can be adapted to various CMP architectures, and hence is also viable for large design space exploration. Details of the simulation tool can be found in [3][9]. The queuing network modeling technique provides closed-form solutions in a large design space, covering various thread scheduling disciplines, memory access mechanisms, and processor organizations (Chapter 3 and [27]). Moreover, the framework for fast program-task-to-core mapping (Chapter 4 and [27]) provide solutions as to how to effectively identify the most desirable program-task-to-core mapping that leads to highest overall system performance.

This dissertation makes the following major contributions. First, it proposes a performance analysis methodology for multithreaded multicore processors, which is able to characterize the general performance properties for a wide variety of CMP architectures and a large workload space at coarse granularity. Second, based on this methodology, it establishes a modeling technique which maps large classes of multithreaded multicore processors to queuing network models with closed-form solutions in a large design space. Third, it develops a novel framework and resulting fast algorithms to enable program-task-to-core mapping that maximizes the overall system throughput/utilization.

The rest of the dissertation is organized as follows. Chapter 2 describes the proposed analysis methodology. Chapter 3 presents the queuing network modeling technique. Chapter 4 proposes the framework for program-task-to-core mapping. Finally, Chapter 5 concludes the dissertation.

CHAPTER 2

METHODOLOGY

2.1 Introduction

The main idea of our methodology is to capture only activities that have major impact on the thread level performance. In other words, the instruction level and microarchitectual details are overlooked, unless they trigger events that may have significant effects 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. Correspondingly, all the components including CPUs, caches, memories, and interconnection networks are modeled at a highly abstract level, overlooking microarchitectual details, just enough to capture the thread level activities. For example, for a CPU running a coarse-grained thread scheduling discipline and a memory with a FIFO queue, they are modeled simply as queuing servers running a coarse-grained thread scheduling algorithm and FIFO discipline, respectively.

The following sections describe, at the thread level, the modeling of the CMP organization, the workload, and the design space, separately. Since in the CMP family, CPs are particularly difficult to model, as explained in Chapter 1, in the rest of this dissertation and without loss of generality, we discuss CMP in the context of CP. All we need to note is that for a CP, a program task mapped to a thread in a core comes from a packet and the packet arrival process and packet mixture (or code path mixture) determining the workload characteristics, rather than a program or program tasks loaded in that core.

2.2 CMP Organization

We consider a generic CP organization depicted in Fig. 2.1. This organization focuses on the characterization of multicore and multithread features common to most of the CMP 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 and/or off-chip supporting components, such as I/O interfaces, memories, level one and level two caches, special processing units, scratch pads, embedded general-purpose 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 may run more than one thread and the threads are scheduled based on a given thread scheduling discipline.

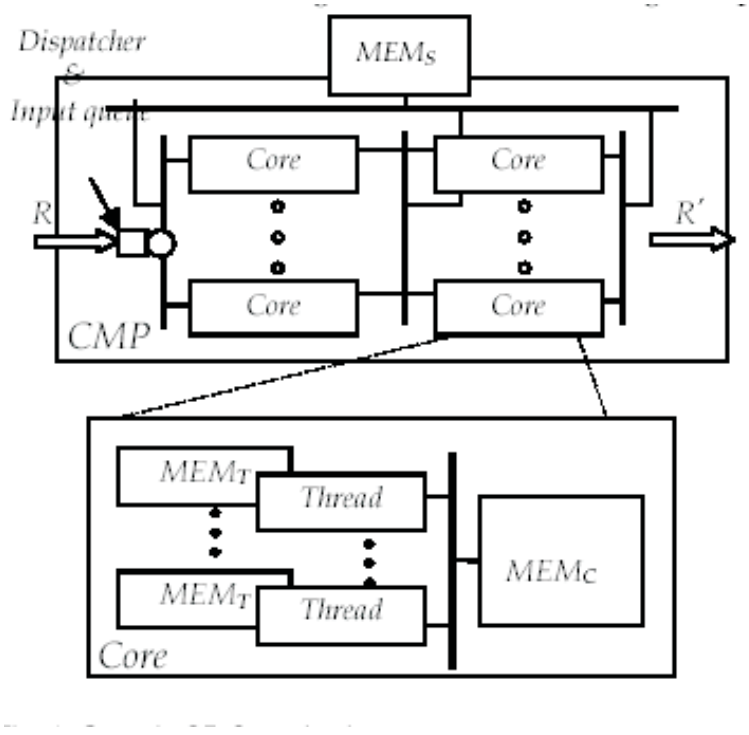

Figure 2.1. CMP Organization.

Cores may be configured in parallel and/or multi-stage pipeline (a two-stage configuration is shown in Fig. 2.1) and there is a packet stream coming in from one side and going out through the other side. Packet processing tasks may be 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.

Clearly, the above organization also applies to non-CP based CMPs. The only difference is that in this case there is no packet arrival or departure processes and tasks mapped to different cores are generated by one or multiple applications mapped to those cores. This dissertation 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 budget.

2.3 Work Load

At the core of our methodology is the modeling of the workload, defined as a mapping of program tasks to threads in different cores, known as code paths. For tasks mapped to a given thread, a piece of pseudo code for those 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 thread. A specific packet processing or instantiation of program execution is associated with a specific code path, or a sequence of events that the thread needs to execute to fulfill the tasks. For a CP, the program tasks mapped to a thread may be on-and-off, which is a function of the packet arrival process. Moreover, what code path a thread may need to handle in a given time period is dependent on the actual mixture of packets of dif-

ferent types arriving in that time period, each being associated with some distinct program tasks to be fulfilled, known as a mixture of code paths. For example, while an IP packet may subject to the entire cycle of both layer 2 and layer 3 processing, resulting in a long code path, a packet carrying IS-IS routing information is forwarded to the control plane immediately after it is identified at layer 2, which leads to a very short code path. In this dissertation, a code path is defined at the thread level, which is composed of a sequence of segments corresponding to different events that have significant impact on the thread-level performance. For each segment, we are only concerned with the segment length in terms of the number of core cycles. It can be formally defined as follows:

$T_k(M_k; m_{1,k}, t_{1,k}, \tau_{1,k}, \cdots, m_{M_k,k}, t_{M_k,k}\tau_{M_k,k})$: 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 $k = 1, \cdots, K$ and $i = 1, 2, \cdots, M_k$; K is the total number of code paths in the pseudo code; and M_k is the total number of events in the code path k.

A graphic representation of such a code path is given in Fig. 2.2.

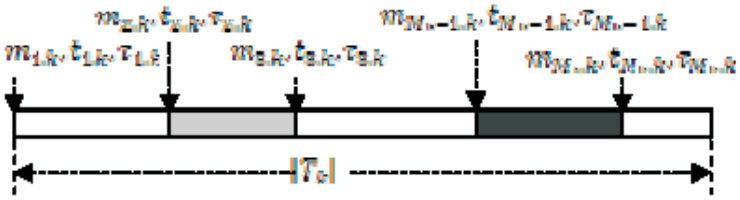

Figure 2.2. $T_k(M_k; m_{1,k}, t_{1,k}, \tau_{1,k}, \cdots, m_{M_k,k}, t_{M_k,k}\tau_{M_k,k})$.

We note that a code path thus defined is simply a sequence of events with event inter-arrival times $t_{i+1,k} - t_{i,k} = \tau_{i,k}$ for $i = 1, 2, \cdots, M_k - 1$. The events $m_{i,k}$ $\in$CPU are represented by the white segments and the corresponding $\tau_{i,k}$ is the number of core cycles the CPU spends on this thread in this segment. All other events are separated by the CPU events. For an event $m_{i,k} \in MEM_T$, MEM_C, or MEM_S, $\tau_{i,k}$ represents the unloaded

resource access latency. An event can be introduced to account for the serialization effect caused by, for example, a critical region. Hence, the length of the code path T_k , denoted as $|T_k|$, is the total duration of code path k handled by a thread in the absence of resource contention, i.e., without waiting times due to contention with other threads for CPU, memory, and any other resource accesses. An event is defined as one that is expected to have a significant impact on the thread-level activities. Currently, we have defined the following four types of events: (1) CPU events; (2) resource access events which may cause significant delay and thread-level interactions (i.e., context switching), i.e., $m_{i,k} \in MEM_T$, MEM_C, or MEM_S; and (3) events that cause serialization effects, e.g., a critical region. More types of events can be incorporated if they are expected to contribute significantly to the thread-level activities.

2.4 Design Space

We want the design space to be as large as possible to encompass as many CMP architectures and workloads as possible. Fig. 2.3 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. Fig. 2.3 also shows the part (i.e., the small cone on the left) that has been (incompletely) explored by the existing work using queuing network modeling techniques (see Section 3.6 for more details). Clearly, the existing work only covers a tiny part of the entire 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 CMP 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 path mixtures, for a given type of processor organization. The number-of-core and number-of-thread-per-core dimensions determine the size of the CMP in terms of the numbers of cores and threads. The number-of-thread-per-core dimension also needs to deal with dynamic multithreading, where the number of threads used for a program task may change over time, due to on-and-off packet arrivals or the variation of the level of parallelism in a program.

In summary, this chapter described a methodology that provides a coarse-granular, thread-level view of a CMP in general and a CP in particular, in terms of its organization and design space. Based on this methodology, the following two chapters demonstrate how an analytical modeling technique and a framework for fast program-task-to-core mapping can be developed to allow much of the design space in Fig. 2.3 to be explored.

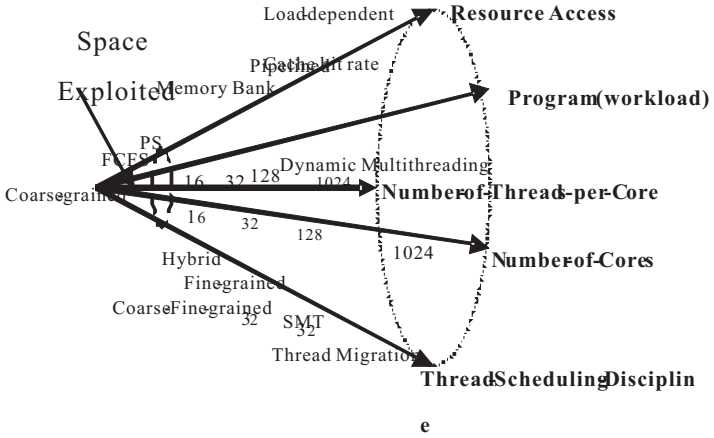

Figure 2.3. Design Space.

2.5 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., [31][38][39]) makes the short synthetic trace from a long real program trace and save time by simulating the short statistic trace. Partial simulation (e.g., [34][35][40]) 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., [32][33][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 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.

CHAPTER 3

ANALYTIC MODELING TECHNIQUE

This chapter is organized as follows. Sections 3.1-3.3 introduces the analytic modeling technique. The technique is then tested against cycle-accurate simulation in Section 3.4. Application of this technique to the bottleneck analysis for multithreaded single-core processors is given in Section 3.5. Finally, the related work is reviewed in Section 3.6.

3.1 Basic Idea

We start with a simple example to bring out the main ideas and then formalize the modeling technique. Consider a code path with one memory access on the left in Fig. 3.1. Now assume the CPU is coarse-grained and the memory is FIFO. Then two active threads loaded with the same code path in Fig. 3.1 have the execution sequences as given on the right in Fig. 3.1. The yellow segments are the thread waiting times. Now, consider a closed queuing network composed of two FIFO queuing servers, modeling a coarse-grained CPU and an FIFO memory, as shown in Fig. 3.2(a). Assume that there are two jobs circulating in this network, modeling the two actively threads.

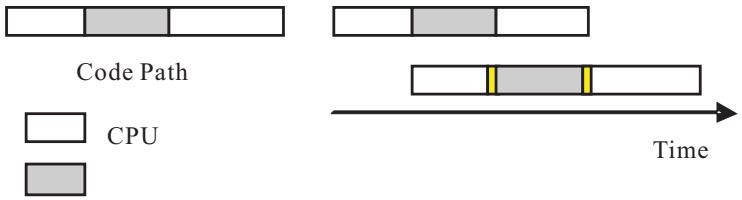

Figure 3.1. Execution Sequence for Coarse-Grained Core.

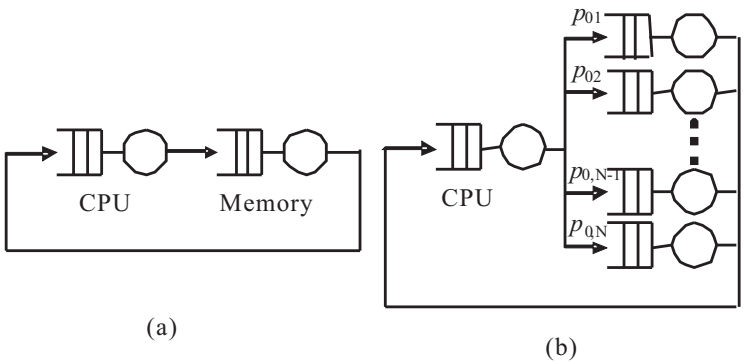

Figure 3.2. Queuing Network Models: (a) Two Components (b) Multiple Components.

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 Fig. 3.1. 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 Fig. 3.1. Again, if the service times exactly match the segment lengths, the thread circulation exactly recovers the execution sequence in Fig. 3.1.

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 [5]). 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 should be used to characterize the service time. More-

over, for a code path that characterizes a workload for a processor with multiple parallel resources, the corresponding closed queuing network, as depicted in Fig. 3.2(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 CMPs with M components and any long-run workloads can be generally modeled as a closed queuing network with M 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 is then 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 wide range of CMP architectures.

3.2 Design Space

The queuing network modeling techniques at our disposal limit the size of the design space to one that must be mathematically tractable. This makes the coverage of the design space in Fig. 2.3 a challenge. In this subsection, we discuss our solutions in meeting the challenge.

Memory/interconnect-network 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 1 shows how individual instances

Table 3.1. Component modeling using queuing models with local balance equations

Component \ Queue Model	$M/G/\infty$	M/M/m FCFS	M/G/1 PS	M/M/1
SMT	✓	✓		
Fine-Grained Thread scheduling			✓	
Coarse-Grained Thread scheduling scheduling				✓
Hybrid-Fine-and-, Coarse-Grained Thread scheduling		✓		
Resources dedicated to individual threads	✓			
FCFS shared Memory, Cache, Interconnection Network,or Creitical Region				✓
FCFS Memory with with Popelined Access		✓		

in these two dimensions can be modeled by three queuing models with local balance equations (according to the BCMP theorem [4]), including $M/G/\infty$; $M/M/mFCFS$ (including $M/M/1$); and $M/G/1PS$ (processor sharing). Note that 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 3.2, 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 3.2:

- 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 $M/G/\infty$ queue, mimicking multiple CPUs handling all the threads in parallel, otherwise, it

can be approximately 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 approximately 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 $M/G/\infty$ 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 modeled as an M/M/1 queue.
- FCFS Memory with Pipelined Access: Memory banks can be accessed in parallel. It can be modeled as an M/M/m FCFS queue, with up to the number-of-bank worth of memory accesses serviced in parallel and the rest queued in a FCFS queue.
- FCFS Memory with Pipelined Access: Same as above. The pipeline depth determines how many threads can be serviced simultaneous in the M/M/m FCFS queue.

We note that the memory/interconnection-network 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 either positively or negatively correlated with the number of threads in use due to temporal locality and cache resource contention. These effects can be accounted for in our framework without approximation, by means of the existing load-dependent routing techniques (e.g. [1]). 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 [5] [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 3.2, 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 our future research will explore more sophisticated queuing models in an attempt to overcome this limitation, we expect that the first order approximation could actually provide good performance data, due to the well-known robustness property [5], which states that for closed queuing networks, the system performance is insensitive to the service time distributions. To calculate ($\{\mu_i\}$, $\{p_{ij}\}$), we first define p^k, the probability that an incoming packet is associated with code path k (for $k = 1, 2, \cdots, K$). In other words, p^k defines a

code path mixture. We further define τ_i^k , f_i^k and q_{ij}^k as the average service time at queuing server i; the frequency to access queuing server i; and the probability to access queuing server j upon exiting queuing server i, respectively, for a thread handling code path k. These statistic parameters are collectable from the pseudo code. Then the average service rates and routing probabilities for a given job class can be written as:

$$\mu_i = \frac{\sum_{k=1}^{K} p^k f_i^k}{\sum_{k=1}^{K} p^k f_i^k \tau_i^k}, \qquad i = 1, \cdots, M \tag{3.1}$$

and

$$P_{ij} = \sum_{k=1}^{K} p^k q_{ij}^k, \qquad i = 1, \cdots, M \tag{3.2}$$

Here a job class is defined as threads that follow these same statistics. In general, all the threads belong to the same core forms a job class.

Number-of-core and Number-of-thread Dimensions: As we shall see in Section 3.3, these dimensions can be fully covered for the queuing server models described in Table 3.2.

3.3 Performance Measures

For CMPs in the design space covered by the queuing server models in Section 3.2, 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,

$$N = \sum_{i=1}^{M} k_i, \qquad k_i = \sum_{r=1}^{R} k_{ir}, \tag{3.3}$$

where k_{ir} is the number of jobs in the rth 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 [4], the state probabilities of the system can be written as:

$$\pi(S_1, \cdots, S_n) = \frac{1}{G(N)} \prod_{i=1}^{M} f_i(S_i), \tag{3.4}$$

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

$$G(N) = \sum_{\sum_{i=1}^{n} S_i = N} \prod_{i=1}^{M} f_i(S_i), \tag{3.5}$$

The $f_i(S_i)$'s are the relative state probabilities at the node i , which are defined as follows:

$$f_i(S_i) = \begin{cases} k_i! \frac{1}{\beta_i(k_i)} \cdot (\frac{1}{\mu_i})^{k_i} \cdot \prod_{r=1}^{R} \frac{1}{k_{ir}!} e_{ir}^{k_i r}, & for - /M/m/ - FCFS \\ k_i! \prod_{r=1}^{R} \frac{1}{k_{ir}!} e_{ir}^{k_i r}, & for - /G/1 - PS\&LCFSPR \\ \prod_{r=1}^{R} \frac{1}{k_{ir}!} e_{ir}^{k_i r}, & for - /G/\infty \end{cases} \tag{3.6}$$

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

$$e_{ir} = \sum_{j=1}^{M} \sum_{s=1}^{R} e_{js} \cdot p_{js,ir}, \qquad for \quad i = 1, \cdots, M, \qquad r = 1, \cdots, R \tag{3.7}$$

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

$$\beta_i(k_i) = \begin{cases} k_i! & k_i \leq m_i \\ m_i! \cdot m_i^{k_i - m_i} & k_i \geq m_i \\ 1 & m_i = 1 \end{cases} \tag{3.8}$$

where m_i is the number of servers in node i. Based on the generation function defined above, relevant performance measures in our model can be written as follows [5]:

Throughput:

$$\lambda_i = \sum_{r=1}^{R} = \sum_{r=1}^{R} e_{ir} \cdot \frac{G(N-1_r)}{G(N)}, \qquad for \quad i = 1, \cdots, M, \tag{3.9}$$

Mean Response Time: By the little's law,

$$T_i = \frac{l_i}{\lambda_i} \tag{3.10}$$

where the mean number of jobs at the ith queuing server l_i is,

$$\begin{aligned} l_i &= \sum_{r=1}^{R} l_i r \\ &= \sum_{r=1}^{R} \sum_{\sum_{j=1}^{M} S_j = N \& S_i = k} k_i r \cdot f_i(K) \cdot \frac{G^{(i)}(N-k)}{G(N)}, \qquad for \quad i = 1, \cdots, M \end{aligned} \tag{3.11}$$

where $G^{(i)}$ 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 (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.

3.4 Testing

In this section, the accuracy for the proposed analytic modeling technique is tested against cycle-accurate simulators (CAS). Since CPs are most difficult to deal with, we test our solution against a cycle-accurate CP simulator, i.e., IXP 1200 SDK Developer workbenches [7]. With a set of code samples available in both IXP1200, the sustainable line rates obtained from our techniques 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. The code samples and corresponding analytic model and simulation setups are described in Section 3.4.1 and the Section 3.4.2 presents the test results.

3.4.1 Test Case

Since all the cores in IXP1200 run a coarse-grained thread scheduling discipline, our analytic model is configured to run the coarse-grained thread scheduling algorithm as well. The program tasks mapped to the cores for IXP1200 sample applications are briefly described as follows.

IXP1200 code samples: Three different code samples, Generic IPv4 Forwarding, Layer-2 Filtering, and ATM/Ethernet IP Forwarding, available in IXP1200 Developer workbench [7] are tested. The worst-case code paths at the bottleneck cores for these code samples are given in Appendix. 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.

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 Fig. 3.3 and ATM/Ethernet Forwarding is mapped to three core pipeline stages as shown in Fig. 3.4. 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.

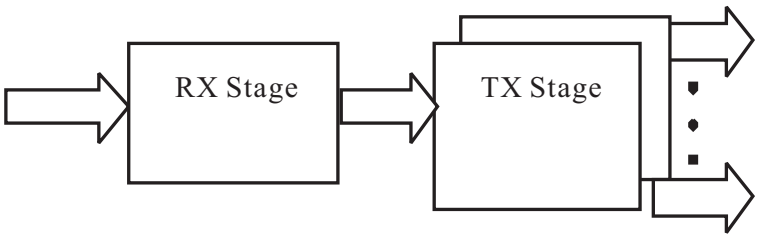

Figure 3.3. Pipeline configuration for Generic IP forwarding and Layer-2 Filtering.

The parameter settings for the simulation are as follows:

IXP1200: ME clock rate = 200/600 MHz

Packet size = 64/64 bytes, DRAM = 24/64MB

SRAM = 1/64 MB (for each channel of two)

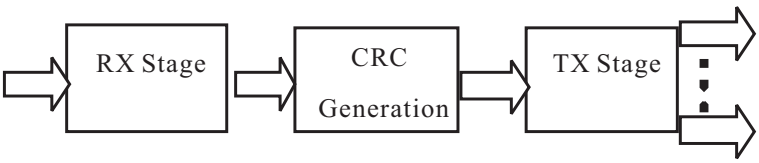

Figure 3.4. Pipeline configuration for ATM/Ethernet Forwarding.

3.4.2 Simulation and Analytic Model Setup

Our analytic model only needs to deal with 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 multistage pipeline. All the worst-case code paths for the corresponding bottleneck cores in table format are listed in Appendix. 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 specific features used in our model. The rest are generic or common features pertaining to all the CP architectures. Clearly, the code paths as given in Appendix can be easily derived from a piece of pseudo code provided by the user.

All the cases studied can be modeled as a single-class, single core system with a coarse-grained CPU and a set of parallel resources including a SRAM, a DRAM, an FBI, a RFIFO, and a scratchpad, as depicted in Fig. 3.5. In our model, we treat all these resources as local to the core and assume that these resources can be accessed in parallel. This is justified by the fact that in IXP1200 simulators, the resource contention for accessing shared resources are not accounted for and the fact that SDRAM and SRAM accesses are

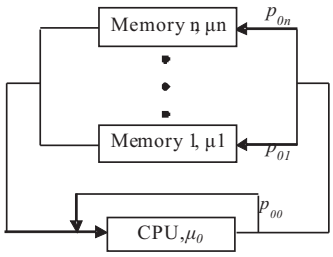

Figure 3.5. Closed Queuing Network Model for Bottleneck Core.

optimized based on multiple memory banks and a separation of read and write operations, respectively.

For analytic modeling, the coarse grained CPU is approximately modeled as an FCFS M/M/1 queue and all the parallel resources as $M/M/\infty$ queue. Also, to study the throughput performance, we assume that all threads in the core are kept busy. As a result, the system is modeled as a closed queuing network consisting of a CPU (M/M/1 queue) and multiple local resources ($M/M/\infty$ queues) with fixed number of jobs as in Fig 3.5. The workload parameters ($\{\mu_i\}$, $\{p_{ij}\}$) are estimated from the code paths based on 3.1 and 3.2.

3.4.3 Results

In this Section, sustainable line rates are obtained from Analytic Method (AM), and those obtained from IXP1200 CAS for the code samples described in previos section. In each table, total latency for a packet, sustainable line speed, and the accuracy of AM against Intel IXP1200 CAS are given in the same format for each code path sample. For IXP 1200 case studies in Table 3.2 to Table 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 AM and CAS, respectively. The last column lists the percentage difference of the sustainable line speeds obtained from AM versus CAS.

Table 3.2. The AM versus CAS(IXP1200) for *Generic IPv4 forwarding*

Number of Threads	Total Latency (cpu cycles)		Line Speed (Mbps)		% line speed error rate against CAS (AM)
	AM	CAS	AM	CAS	
1	523	537	196	191	2.62
2	599	600	342	341	0.29
3	705	687	436	447	2.46
4	846	876	484	467	3.64

Table 3.3. The AM versus CAS(IXP1200) for *ATM/Ethenet IP Forwading*

Number of Threads	Total Latency (cpu cycles)		Line Speed (Mbps)		% line speed error rate against CAS (AM)
	AM	CAS	AM	CAS	
1	741	724	138	141	2.13
2	844	812	243	252	3.57
3	988	981	311	313	0.32
4	1180	1184	347	346	0.29

Table 3.4. The AM versus CAS(IXP1200) for *Layer-2 Filtering*

Number of Threads	Total Latency (cpu cycles)		Line Speed (Mbps)		% line speed error rate against CAS (AM)
	AM	CAS	AM	CAS	
1	709	730	144	140	2.86
2	795	798	258	257	0.39
3	915	978	338	314	7.64
4	1076	1157	381	354	7.63

For all the cases studied, the results obtained from AM are within 8% of the CAS results. Moreover, the results for Am are obtained in subseconds on a Pentium IV PC.

3.5 Bottleneck Identification for Sigle Core Processors

3.5.1 Model

In this section, we consider a class of queuing network models given in Fig. 3.6. This class of models characterizes a class of processors with a single CPU, a cache, and an arbitrary number of parallel resources, denoted as m_q, (e.g., a main memory, a coprocessor, an I/O device, or even a critical region). In this model, we assume that the interconnection network has sufficient bandwidth to transfer data between CPU and the parallel resources without creating a bottleneck and hence it is not explicitly modeled. Upon exiting the CPU server, a thread has probability p_{0i} to visit parallel resource i, where $i = 1, 2, \cdots, m_q$. In the case of memory resource access, the thread will first check if the requested data is available in the cache. In our model, no details of cache access mechanisms are modeled,

except a cache hit probability $P_{ih}(S_i)$ that causes the thread to immediately loop back to the CPU and a cache miss probability $(1 - P_{ih}(S_i))$ that causes the thread to access the ith resource. Here S_i is the size of the cache memory block allocated to the cached data from memory resource i. All the other components are modeled as queuing servers. The CPU queuing server mimics the thread scheduling disciplines listed in the thread scheduling discipline dimension in Fig. 2.3 and all other parallel queuing servers mimic the resource access mechanisms in the resource dimension in Fig. 2.3. An arbitrary number of threads, M_t, circulates in the closed queuing network, with routing probability p_{ij} to visit server j upon exiting server i.

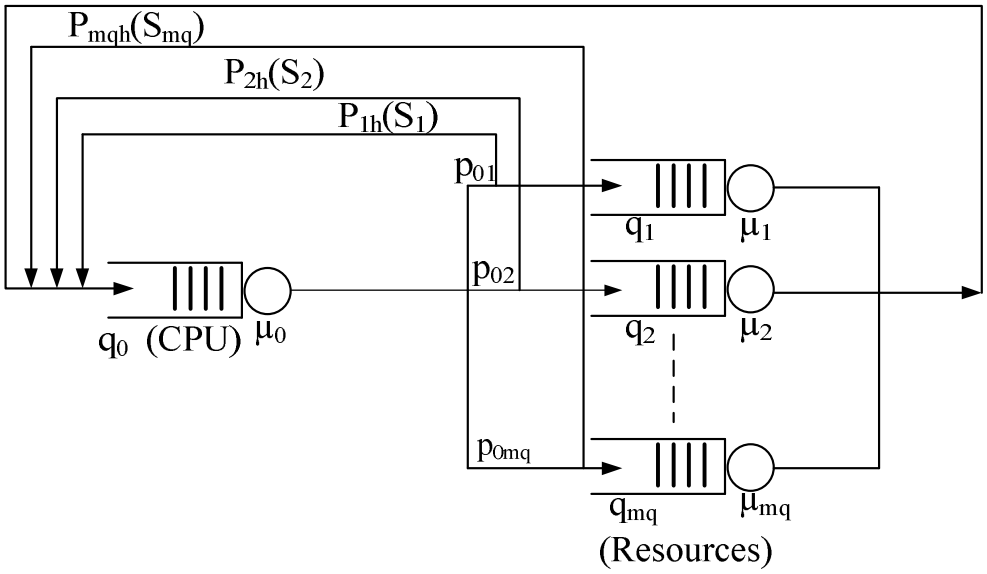

Figure 3.6. Single Core Queuing Network Model.

Component Models: Without resorting to any approximation techniques, the existing queuing network modeling techniques will allow both resource and thread scheduling discipline dimensions in the design space (see Fig. 2.3) to be exploited analytically. Any

instance in these two dimensions can be modeled using a queuing model that has local balance equations (i.e., it leads to solutions of product form or closed form). More specifically, Table 3.2 shows how these two dimensions can be modeled by only three queuing models with local balance equations, including $M/G/\infty$; M/M/m FCFS (including M/M/1); and M/G/1 PS (processor sharing).

3.5.2 Main Results

To limit the exposure, for the time being, we assume that there is no cache in the model in Fig. 3.6, i.e., $P_{ih}(S_i) = 0$. The results with caching will be given at the end of this section. We focus on the performance measure $P_I(m)$, i.e., the probability that there are m threads at the CPU queuing server. In particular, we are interested in its asymptotic behavior, i.e., whether $\lim_{M_t \to \infty} P_I(0) = 0$. It tells us whether or not multithreading can completely hide the resource access latencies from the CPU, provided that the thread resource is abundant, and hence, whether the multithreading can help achieve the maximum throughput performance. This performance measure will lead to the identification of the general conditions under which the bottleneck resources are bound to appear, regardless how many threads are used.

Define $f_i(k_i)$ to be the steady state probability that there are k_i threads at queuing server i, for $i = 0, 1, \cdots, m_q$, where $\sum_{i=0}^{m_q} k_i = M_t$. Let q_i represent queuing server i (see Fig. 3.6), for $i = 0, 1, \cdots, m_q$. Following the convolution algorithm [30], we have,

$$P_I(0) = \frac{f_0(0) q_{1*2*\cdots*m_q}(M_t)}{q_{0*1*2*\cdots*m_q}(M_t)} \tag{3.12}$$

where

$$q_{0*1*2*\cdots*m_q}(M_t) = \sum_{m+n=M_t} f_0(m) q_{1*2*\cdots*m_q}(n) \tag{3.13}$$

and

$$q_{1*2*\cdots*m_q}(n) = \sum_{k_1+\cdots+k_{m_q}=n} \prod_{i=1}^{m_q} f_i(k_i) \tag{3.14}$$

Eq. (3.12) holds true for any queuing servers listed in Table 3.2 and any model parameters ($\{\mu_i\}, \{p_{ij}\}$) . In other words, $P_I(0)$ is a performance measure for a class of processor models defined in the entire design space in Fig. 2.3.

According to Table 3.2, both $M/G/\infty$ and $M/M/m$ queuing models can be used to model SMT-based CPU server and the $M/G/1$ PS queuing model is used to model fine-grained CPU server in Fig. 3.6. All the resource-related servers can be modeled using $M/M/1$ and $M/M/m$ queuing models. To simplify the design, in this paper, we adopt $M/M/m$ for SMT and only consider a special case of $M/G/1$ PS, i.e., $M/M/1$ PS for the fine-grained CPU server. With these simplifications, the service time distributions for all the queuing servers are then exponentially distributed and uniquely determined by their average service rates μ_i for $i = 0, 1, \cdots, m_q$ and $f_i(k_i)$ can be generally express as follows:

$$f_i(k_i) = \frac{\alpha_i^{k_i}}{\beta_i(k_i)} \tag{3.15}$$

where α_i is the relative utilization of q_i. $\alpha_i = \frac{p_{0i}e_i}{\mu_i}$ for $i = 1, 2, \cdots m_q$ and $\alpha_0 = \frac{e_0}{\mu_0}$ and e_i is the relative thread arrival rate at q_i. In this system, $e_0 = \sum_{i=1:m_q} e_i$. $\beta(x)$ is define as follows:

$$\beta(x) = \begin{cases} 1 & M/M/1 \quad \& \quad M/M/1\ PS \\ x! & M/M/\infty \\ x! & M/M/m \quad FCFS \quad x \leq m \\ m!m^{(x-m)} & M/M/m \quad FCFS \quad x > m \end{cases} \tag{3.16}$$

Substituting Eq.(3.15) into Eq. (3.12), we have,

$$P_I(0) = \frac{\sum_{k_1+\cdots+k_{m_q}=M_t} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)}}{\sum_{n=0}^{M_t} \frac{1}{\beta_0(M_t-n)} \sum_{k_1+\cdots+k_{m_q}=n} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)}} \tag{3.17}$$

where $a_i = \frac{p_{0i}\mu_0}{\mu_i}$.

We now have the following general result (see Appendix B for detailed proof):

Theorem : $\lim_{M_t\to\infty} P_I(0) = 0$ if and only if $\frac{m_0 a_i}{m_i} \leq 1, \forall i = 1, 2, \cdots, m_q$.

The Theorem simply states that multithreading can completely hide the resource access latencies from the CPU, if only if $m_0 a_i/m_i < 1$ for all the resources. In other words, resource i is identified as a bottleneck if $m_0 a_i/m_i > 1$. To remove a bottleneck resource i, one needs to increase the relative service rate of the resource (i.e., μ_i/μ_0), reduce the resource access frequency p_{0i}, or increase the relative parallelism or deepen the pipeline for the resource access (i.e.,m_i/m_0), to the extend that $m_0 a_i/m_i$ is reduced to be less than one. As a result, this Theorem quantitatively characterizes the general conditions under which bottleneck resources appear.

So far, we have assumed that there is no caching effect, i.e., $P_{ih}(S_i) = 0$, for $i = 1, 2, \cdots, m_q$. Since S_i is the cache resource allocated to accommodate the cached data from resource i, we must have $S \geq \sum_{i=1:m_q} S_i$, where S is the total cache size. Now, we take into account of the caching effect for the model in Fig. 3.6, i.e., $P_{ih}(S_i) \geq 0$. To simplify the discussion, we assume that all the resources are memory resources, so that caching can help reduce the resource access latencies for all the resources.

Assuming there is no correlation among consecutive cache hits, our cache model only amounts to the change of p_{0i} to $(1 - P_{ih}(S_i))p_{0i}$ and consequently, a_i changes to $(1 - P_{ih}(S_i))a_i$. The product-form property of the model is preserved. Hence, we have the

following corollary in parallel to Theorem,

Corollary : $\lim_{M_t \to \infty} P_I(0) = 0$ if and only if $\frac{m_0(1-P_{ih}(S_i))a_i}{m_i} \leq 1, \forall i = 1, 2, \cdots, m_q$.

To illustrate the power of and the intuition behind the above results, let us walk through a special case. Consider the case when $m_q = 1$, i.e., the model in Fig. 3.6 only has two queuing servers, a CPU server and a resource server and $P_{1h}(S_1) = 0$. Following the same procedure that leads to Eq. (3.17), we have,

$$q_{0*1}(M_t) = \sum_{k_0+k_1=M_t} f_0(k_0) f_1(k_1) \tag{3.18}$$

and

$$P_I(0) = \frac{f_0(0) f_1(M_t)}{\sum_{k_0+k_1=M_t} f_0(k_0) f_1(k_1)} \tag{3.19}$$

Since $f_i(0) = 1$, Eq. (3.19) can be simplified as:

$$P_I(0) = \frac{f_1(M_t)}{\sum_{k_0+k_1=M_t} f_0(k_0) f_1(k_1)} \tag{3.20}$$

By substituting Eq. (3.15) into Eq. (3.20), we have

$$P_I(0) = \frac{\alpha_1^{M_t}/\beta_1(M_t)}{\sum_{k_1=0}^{M_t} \frac{\alpha_0^{M_t-k_1}}{\beta_0(M_t-k_1)} \frac{\alpha_1^{k_1}}{\beta_1(k_1)}} \tag{3.21}$$

Define $a_i = \frac{\mu_0}{\mu_i}$, then $\alpha_i = \alpha_0 a_i$. Eq. (3.21) can be reduced to:

$$P_I(0) = \frac{a_1^{M_t}/\beta_1(M_t)}{\sum_{k_1=0}^{M_t} \frac{a_1^{k_1}}{\beta_0(M_t-k_1)\beta_1(k_1)}} \tag{3.22}$$

Substituting $\beta(x)$ function into Eq. (3.22) and with some rearrangement, we get,

$$P_I(0) = \frac{\frac{m_1^{m_1}}{m_1!}\left(\frac{a_1}{m_1}\right)^{M_t}}{\frac{m_0^{m_0}}{m_0! m_0^{M_t}} \sum_{k_1=0}^{m_1} \frac{a_1^{k_1}}{k_1! / m_0^{k_1}} + \frac{m_0^{m_0} m_1^{m_1}}{m_0! m_1! m_0^{M_t}} \sum_{k_1=m_1+1}^{M_t-m_0} \frac{a_1^{k_1}}{\left(\frac{m_1}{m_0}\right)^{k_1}} + \frac{m_1^{m_1}}{m_1!} \sum_{k_1=M_t-m_0+1}^{M_t} \frac{a_1^{k_1}}{(M_t-k_1)! m_1^{k_1}}} \tag{3.23}$$

Note the second term in the denominator is a geometric progression whose sum is given by:

$$\sum_{k_1=m_1+1}^{M_t-m_0} \frac{a_1^{k_1}}{\left(\frac{m_1}{m_0}\right)^{k_1}} = \frac{\left(\frac{a_1 m_0}{m_1}\right)^{m_1+1} - \left(\frac{a_1 m_0}{m_1}\right)^{M_t-m_0+1}}{1-\frac{a_1 m_0}{m_1}} \tag{3.24}$$

Let $G = \frac{a_1 m_0}{m_1}$ and substituting Eq. (3.24) into Eq. (3.23), we have

$$P_I(0) = \frac{1}{\frac{m_0^{m_0} m_1!}{m_1^{m_1} m_0!} \sum_{k_1=0}^{m_1} \frac{a_1^{k_1}}{k_1!/m_0^{k_1}} G^{-M_t} + \frac{m_0^{m_0}}{m_0!} \frac{G^{(m_1+1-M_t)} - G^{-m_0+1}}{1-G} + \sum_{k_1=M_t-m_0+1}^{M_t} \frac{a_1^{k_1} m_0^{M_t}}{(M_t-k_1)! m_1^{k_1}} G^{-M_t}} \tag{3.25}$$

As M_t goes to infinity, we have,

$$\lim_{M_t \to \infty} P_I(0) = \begin{cases} \frac{1}{\frac{m_0^{m_0}}{m_0!}\frac{G^{-m_0+1}}{G-1} + \sum_{k'=0}^{m_0-1}\left(\frac{a_1}{m_1}\right)^{-k'}/k'!} & G > 1 \\ 0 & G < 1 \end{cases} \tag{3.26}$$

This gives the general condition $G > 1$ under which the resource becomes a bottleneck, where $G = \frac{a_1 m_0}{m_1} = \frac{\mu_0 m_0}{\mu_1 m_1}$, agreeing with the general result in Theorem. This condition tells us that if the average service rate times the level of parallelism (i.e., m_1) at the resource is slower than the service rate times the level of parallelism at the CPU server, the resource becomes a bottleneck that throttles the overall throughput.

Finally, it is interesting to note that when $m_0 = m_1 = 1$ (i.e., the CPU is coarse-grained and resource access mechanism is FCFS), we have,

$$\lim_{M_t \to \infty} P_I(0) = \begin{cases} \frac{a_1-1}{a_1} & a_1 > 1 \\ 0 & a_1 < 1 \end{cases} \tag{3.27}$$

A deterministic version of this result was derived in [10] and later a result identical to the one in Eq. (3.27) was derived and studied in [23]. Clearly, the results given in Theorem

are far more general than the results given in [10] and [23]. The proof of of main results is presented in Appendix B.

3.5.3 Thread and Cache Resource Provisioning

With system configuration and workload unchanged, it is clear that cache is needed in addition to multithreading to remove the bottleneck resource i if $\frac{m_0 a_i}{m_i} > 1$, according to Theorem. Now according to Corollary, the minimum amount of cache resource S_i that is needed to remove the bottleneck resource i must satisfy the following equation:

$$\frac{m_0(1 - P_{ih}(s_i))a_i}{m_i} = 1 \tag{3.28}$$

from Eq. (3.28) we have,

$$S_i = P_{ih}^{-1}(1 - \frac{m_i}{m_0 a_i}) \tag{3.29}$$

where P_{ih}^{-1} is the inverse function of P_{ih}. Clearly, $S_i = 0$ if resource i is not a bottleneck resource. The condition that $S \geq \sum_{i=1:m_q} S_i$ gives us a good idea as to how much total cache resource is needed to maximize the throughput performance. If S is a given, this condition determines whether maximum throughput performance can be achieved or not, with maximal thread and cache resource provisioned.

In summary, we have the following generic algorithm for effective thread and cache resource provisioning:

- if $\frac{m_0 a_i}{m_i} < 1$, for all $i = 1, 2, \cdots, m_q$, the maximum throughput performance can be achieved by adding sufficient number of threads and cache is not needed
- else if (without loss of generality) $\frac{m_0 a_1}{m_1} \leq \frac{m_0 a_2}{m_2} \leq \cdots \leq \frac{m_0 a_{k-1}}{m_{k-1}} \leq 1 < \frac{m_0 a_k}{m_k} \leq \cdots \leq \frac{m_0 a_{m_q}}{m_{m_q}}$, calculate S_i for $i = k, \cdots, m_q$ from Eq. (3.29). If $S \geq \sum_{i=k}^{m_q} S_i$, output S_i for cache resource provisioning; else output S_i and request for additional $(\sum_{i=k}^{m_q} S_i - S)$ cache memory.

To make the above discussion generally applicable to any elaborated caching models, so far we have not mentioned what $P_{ih}(S_i)$ should look like. In practice, $P_{ih}(S_i)$ is a complicated function of not only S_i, but also data request patterns, thread scheduling discipline, cache replacement algorithm, etc. Nevertheless, a widely adopted analytical model is: $P_{ih}(S_i) = 1 - \left(\frac{S_i}{\delta} + 1\right)^{-(\epsilon-1)}$, as discussed in [8]. Since how to model the cache hit probability is not the focus of this dissertation, we shall not discuss this issue further in this dissertation.

3.6 Related Work

In terms of queuing network modeling, since Jackson's seminal work [11] 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., [5], [6], [12], [13]). As a result, in the past few decades, queuing networks were widely adopted in modeling computer systems and networks (e.g., [14], [15], [16], [17], [18]).

However, very few analytical results are available for multicore processor analysis. In [19], 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 [20]. Markovian Models are employed in [21] 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 [22]. 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 [17]. 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 crossbar 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 Fig. 2.3). 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 the design space in Fig. 2.3 and that allows system classes to be analyzed over the entire space.

CHAPTER 4

A FRAMEWORK FOR FAST PROGRAM-TASK-TO-CORE MAPPING

Based on the thread-level modeling methodology introduced in Chapter 2, in this chapter, we develop a framework to effectively mapping program-task-to-core to achieve overall high throughput/utilization performance for CMPs. This framework is based on the priorous results in [3][26][27] and the operational analysis of queuing systems in [28]. In this framework, CMPs are modeled at the thread level, following the methodology in Chapter 2 and [27]. Then an iterative procedure is proposed to decouple a many-core system into single core systems, which is solved efficiently using the simulation tool in [3]. This iterative procedure borrows some ideas from the iterative procedure in [25]. However, unlike the procedure in [25], which is subject to the assumptions made in queuing theory, such as stationary and Markovian assumptions, the procedure proposed in this paper applies to any queuing network systems in any given time interval the system is studied. This is made possible by employing simulation tool in [25], rather than numerical analysis of queuing models, to solve the single core problem and the application of operational analysis in [29], rather than queuing theory, to identify the bottleneck resources. With the framework developed, we further design algorithms that allow quick program-task-to-core mapping to maximize overall system throughput/utilization for CMPs with virtually unlimited numbers of cores and threads.

4.1 Introduction

In our framework, first, we assume that a CMP under consideration has N cores sharing a common resource (e.g., a shared memory), and each core has K-1 local resources, a

CPU, and a local cache with negligible service time and cache hit frequency q_{00} as depicted in Fig. 4.1. The CPU and local resources may be different from core to core, running different thread scheduling algorithms, having different resource access mechanisms/bandwidths, and different amounts of resources. Second, we assume that there are a number of types of program tasks with distinct workload characteristics in terms of ($\{\mu_i\}$, $\{q_{ij}\}$), to be mapped to different cores. We further assume that no more than one type of program tasks is allowed to be mapped to any given core and the intensity of the workload on a given core can be increased by adding more threads handling the same type of program tasks.

For a given mapping and by increasing the workload intensity in each core, there are two possible outcomes. One outcome is that all the CPU utilizations reach one. In this case, the overall system throughput/utilization is maximized. The other outcome is that the CMP fails to achieve its maximal overall system throughput/utilization, regardless the workload intensities. This case implies that for the given mapping, there exists at least one bottleneck resource, which prevents at least one CPU from reaching its full utilization. Whether a CMP can achieve its maximal throughput/utilization or not depends on the types of program tasks to be mapped, the types of cores in the CMP modeled in Fig. 4.1, the mapping choice, and the way the workload intensity is assigned to each core. In our framework, we are interested in how to map program-task-to-core and how to assign the workload intensities to achieve the highest possible throughput/utilization for any given types of cores in the CMP in Fig. 4.1 and given types of program tasks. In our framework, we propose a two-step heuristic for program-task-to-core mapping and workload intensity assignment, as discussed separately in the following two sections.

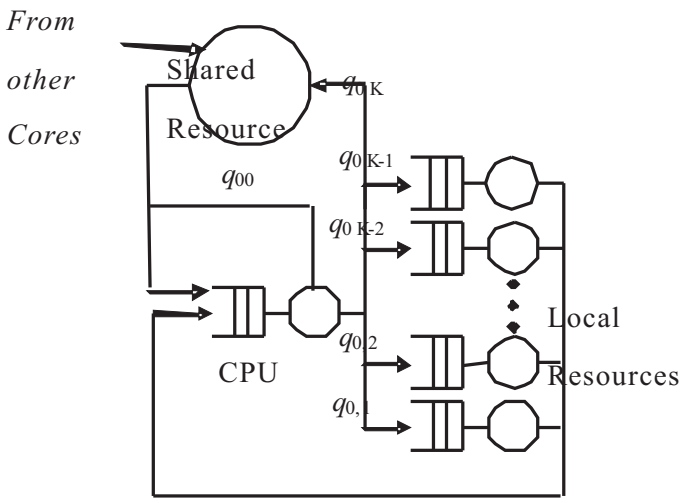

Figure 4.1. CMP Model at Thread Level.

4.2 Program-Task-to-Core Mapping

As our design objective is to maximize the throughput/utilization, we want to map program-task-to-core in such a way that local resources do not pose potential bottlenecks that prevent the CPU from being able to maximize the utilization of its processing power. In the meantime, we want the local resource utilizations to be as close to the CPU utilization as possible. This will ensure that as workload intensity increases, both CPU and all the local resources are more or less fully utilized. Note that in this step, we assume that the shared resource will not pose a potential bottleneck. However, before formalizing this step, we first need to know how to estimate the utilizations for the CPU and local resources for any given type of program-tasks mapped to the core. In general, utilizations are complex functions of workload and system parameters. For the single-core case, however, we now show that the ratios of utilizations can be estimated easily. First we make the following assumption: the workload parameters ($\{\mu_i\}$, $\{q_{ij}\}$) are invariant as the workload intensity increases (i.e., the number of jobs or threads in the system increases). This assumption

holds, since we allow no more than one type of program tasks to be mapped to any given core.

In Chapter 3.5, we were able to show that for a closed-queuing network as in Fig. 4.1 in the absence of other cores sharing the shared resource, which has local balance equations or product-form solutions, the conditions, under which the maximal throughput/utilization can be achieved, are the following (according to Theorem in Section 3.5),

$$\frac{q_{0i} m_0 \mu_0}{m_i \mu_i} < 1, \qquad \forall i, for \quad i = 1, \cdots, K-1 \tag{4.1}$$

The parameters in Eq. (4.1) are either known system parameters (i.e., m_i, the number of parallel servers or the pipeline depth at the queuing server i) or workload parameters measurable from the code paths loaded in the core, i.e., ($\{\mu_i\}$, $\{q_{ij}\}$). This makes it possible to quickly test the performance of any given program-task-to-core mapping, with estimation of just several workload parameters. Nevertheless, the assumption that the close-queuing network must have local balance equations or product-form solutions limits the applicability of this result to a few queuing server models only. Moreover, some key underlying assumptions made in the queuing theory, such as stationary and Markovian properties may not hold true in practice.

As an important part of our framework, we now show that in fact, the conditions in Eq. (4.1) hold true in general, free from all the above assumptions, except the invariance assumption for ($\{\mu_i\}$, $\{q_{ij}\}$). The approach we take follows the operational analysis of the queuing systems developed by Buzen (see [29] and references therein). The operational analysis establishes relationships among system variables (e.g., utilization and throughput) through some basic operationally measurable quantities (e.g., the length of the observation period, the number of jobs arrived during the observation period, and the job finished during the observation period). The assumptions made in queuing theory that are either unrealistic

(e.g., Markovian) or difficult to verify (e.g., steady state and the existence of a well-defined underlying distribution for a stochastic process) are eliminated. Hence, the results derived from this approach can generally be applied to solve real-world problems.

According to [29], for a single-class queuing network (i.e., all the jobs share the same workload parameters ($\{\mu_i\}$, $\{q_{ij}\}$)) and given that the service time S_i (i.e., μ_i^{-1}) and visiting ratio V_i are invariant as the number of jobs increases, the CPU is the bottleneck, if and only if

$$\frac{U_0}{U_i} < 1, \qquad \forall i, for \quad i = 1, \cdots, K \tag{4.2}$$

or equivalently,

$$\frac{V_i S_i}{V_1 S_1} < 1, \qquad \forall i, for \quad i = 1, \cdots, K \tag{4.3}$$

Now, we show that when the job flows are balanced, Eq. (4.3) degenerates to Eq. (4.1) for the queuing network in Fig. 4.1, in the absence of other cores sharing the shared resource (this guarantees that the core involves only a single job class). First, we note that the service ratios satisfy the following balance equations when the job flows are balanced:

$$\begin{aligned} V_0 &= 1 \\ V_j &= q_{0j} + \sum_{i=1}^{K} V_i q_{ij}, \qquad for \quad j = 1, \cdots, M \end{aligned} \tag{4.4}$$

For the single-job-class core given in Fig. 4.1, we have,

$$\begin{aligned} V_1 &= \frac{1}{q_{10}} \\ V_i &= \frac{q_{1i}}{q_{10}}, \qquad for \quad i = 2, \cdots, M \end{aligned} \tag{4.5}$$

We also have,

$$S_i = \frac{1}{m_i}\mu_i, \tag{4.6}$$

where m_i represents the width of parallelism (e.g., the maximum number of issues per cycle at a SMT CPU) or depth of pipeline (e.g., the depth for pipelined memory access) and the service rate at queuing server i. Combining the results in Eqs. (4.3), (4.5) and (4.6), we arrive at Eq. (4.1).

The parameters in Eq. (4.1) are either known system parameters (i.e., m_i) or parameters measurable from the code paths loaded in the core (i.e., Eqs. (3.1) and (3.2)).

With the above preparation, now we can formally state our algorithm in step one as follows:

1. For each type of program-tasks j (for $j = 1, 2, \cdots, M_t$), calculate the ratios $\frac{q_{0i}m_0\mu_0}{m_i\mu_i}$ for all the local resources in all the cores, where M_t is the number of types of program-tasks to be mapped.

2. Select all the cores for which the conditions in Eq. (4.1) hold (for simplicitly, assume there is at least one core that satisfies the conditions). Then rank these cores with increasing order in $\min(\frac{q_{0i}m_0\mu_0}{m_i\mu_i}|_{i=1:K-1})$. Do the same for all types of program-tasks.

3. The type of program-tasks, which has the largest $\min(\frac{q_{0i}m_0\mu_0}{m_i\mu_i}|_{i=1:K-1})$ value among all types of program-tasks will be assigned to the core correspond to this value;

4. Remove the assigned type of program-tasks and the corresponding core from the rank list. Go back to step 3 until all the cores are assigned a type of program-tasks.

Steps 1 and 2 ensure that the local resources will not pose potential bottlenecks for the CPU in any core selected. Steps 3 and 4 guarantee that the type of program-tasks mapped to a core will be the one that makes best use of the local resources.

4.3 Testing of General Conditions

Before discussing step two in the next section, in this section we demonstrate, based on simulation, that the conditions in Eq. (4.1) indeed hold true in general.

Consider a rather extreme case, i.e., the service time distributions for both CPU and memory components are long tailed. More specifically, the Pareto distributions are used to characterize the service times. Pareto distributions account for a wide range of code segment sizes, or equivalently, the thread service times at the CPU, and large variations of memory access latencies. The aim is to test whether such significant deviations from the exponential distributions would (a) shift the appearance of a bottleneck resource away from the point in the parameter space identified by the general conditions; and (b) significantly blur the boundaries between the bottleneck and non-bottleneck regions. We use the simulation results of the original queuing network models as the benchmarks for the testing.

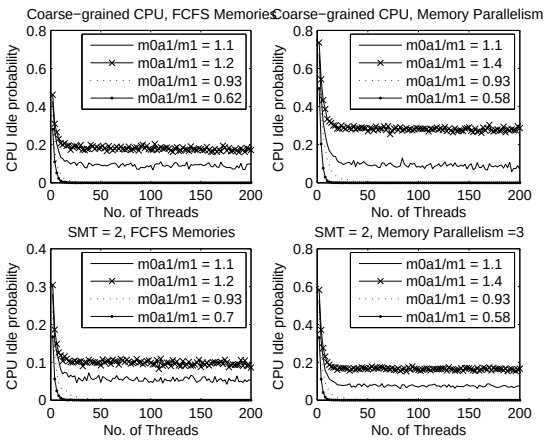

Figure 4.2. Exponential Distribution.

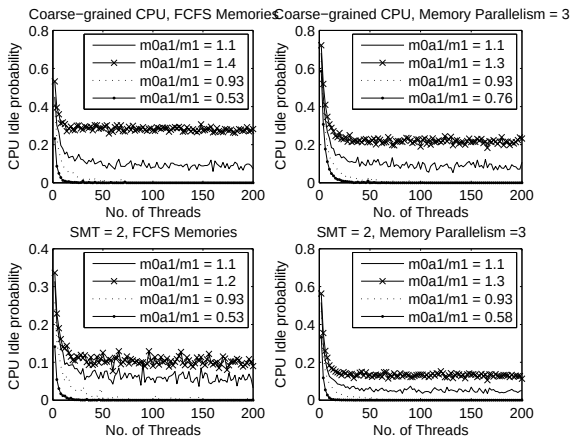

Figure 4.3. Pareto Distribution.

We consider the processor model with four memory resources. We run simulation for both the original queuing network models (whose service times are Exponential) and

the queuing network models with Pareto service times. The results in term of CPU idle probability versus the number of threads are presented in Fig. 4.2 and Fig. 4.3. For each of these two cases, four scenarios are studied: (a) coarse-grained CPU and FCFS memories; (b) coarse-grained CPU and memories with parallel/pipelined accesses; (c) SMT and FCFS memories; and (d) SMT and memories with parallel/pipelined accesses. The result for the four scenarios are presented in the four subplots in Fig. 4.2 and Fig. 4.3. In each subplot, four curves are given. Of which two correspond to the cases where one bottleneck resource is identified according to the general conditions ($\frac{m_0 a_1}{m_1} > 1$), whereas the other two do not involve bottleneck resource according to the general conditions.

As one can see, for both Fig. 4.2 and Fig. 4.3, there is a clean division between the two sets of curves for all the subplots. Namely, as the number of threads increases, the two curves corresponding to the cases without bottleneck resource identified converge to zero, whereas the other two level out at some nonzero values.

The above results clearly indicate that the general conditions derived in Eq. (4.1) is indeed accurate. Hence, step one given in Section 4.2 is practically very useful for effective program-task-to-core mapping for CMPs with many cores.

4.4 Workload Intensity Assignment

Note that whether the shared resource will pose a potential bottleneck for any given core is determined by the workload intensities of all the cores. When the workload intensities for all the cores are low, the shared resource should not pose a potential bottleneck for any core, otherwise more shared resource capacity should be provisioned. This is the reason why we didn't consider this resource in step one, where the program-task-to-core mapping is done, independent of workload intensities, thanks to the invariance of the workload parameters ($\{\mu_i\}$, $\{q_{ij}\}$)'s with respect to the workload intensities.

As the workload intensities increase, the overall system throughput/utilization will increase, until the shared resource becomes overloaded, when it starts to throttle the CPU utilizations for some cores. Note that step one guarantees that the local resources will not pose potential bottleneck for the CPU in any core as workload intensities increase. Hence, the second step of our heuristic is to ensure that the workload intensities assigned to different cores will lead to the overall highest throughput/utilization, while not making the shared resource a bottleneck. In other words, any further increase of the workload intensity for any core will make the shared resource a bottleneck for at least one core.

We first define an important quantity, called utilization increment ratio, $\frac{\Delta U_k}{\Delta U_0}$, where ΔU_k and ΔU_0 are the shared resource and CPU utilization gains, respectively, per workload intensity increase for the core corresponding to the CPU. The unit for the workload intensity can be defined as one thread. Now the idea is to, starting from some minimally required workload intensity for each core, incrementally adding workload intensities, multiple units at a time. Each time, the multiple units is added to the core with the smallest utilization incremental ratio. Ideally, this ratio needs to be updated every time the workload intensity is incremented. The rationale of this approach is to give more workload intensities to cores with less demand of the shared resource, as a way to maximize the overall throughput/utilization gain with minimal shared resource utilization gain. Step two of our heuristic will basically follow this approach with some modifications to reduce the computational complexity. Before formalizing step two, however, we need to know how to calculate utilization incremental ratios and what computational complexity involved.

The key difficulty here is that we are dealing with a CMP with all the cores coupled together through the shared resource. In our previous work [25], we demonstrated that the cores can be decoupled through an iterative procedure. To be mathematically tractable, however, CMPs were modeled as closed queuing networks with closed-form solutions. In this dissertation, we propose an iterative procedure to decouple the cores, similar to the one

in [25]. However, to avoid the assumptions made in [28], instead of performing numerical analysis in each iteration based on a closed-form solution for each core, the current solution performs simulation analysis using the simulation tool in [25] in each iteration. We first introduce the iterative procedure and then discuss the computational complexity involved.

A key intuition underlying the iterative procedure 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 individual cores can be evaluated quickly as if they were stand-alone ones.

Initially, we calculate the core sojourn time $T_i(0)$ (excluding the shared source) and throughput $\lambda_i(0)$ for single core system i consisting of a single core and the shared resource (for $i = 1, \cdots, N_c$). Then the initial mean sojourn time for all the cores, $T^\star(0)$, is calculated based on the following iteration formulae:

$$T^\star(n) = \sum_{i=1}^{N_c} \frac{\lambda_i(n)}{\sum_{j=1}^{N_c} \lambda_j(n)} T_i(n) \tag{4.7}$$

Then we enter an iteration loop as shown in Fig. 4.4. At the n-th iteration, first the average sojourn time for the shared resource, $T_m(n)$, is calculated based on a two-server queuing network (on the left of Fig. 4.4), including a queuing server for the common memory and an $M/M/\infty$ queuing server characterized by the mean service time $T^\star(n)$. There are a total of $M = \sum_{i=1}^{N_c} m_i$ threads circulating in this network, where m_i is the number of active threads in core i. In other words, we approximate the aggregate effect of

all the threads from all the cores on the common memory using a single $M/M/\infty$ queuing server with the mean service time $T^{\star}(n)$. Then, we test if $|T_m(N) - T_m(N-1)| < \varepsilon$ holds, for 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 $i = 1, \cdots, N$) are updated based on the closed queuing network on the right of Fig. 4.4. This time, the effects of other cores on core i is approximated by a single $M/M/m_i$ server with the mean service time $T_m(n)$. There are m_i threads circulating in this network. Finally, $T^{\star}(n)$ will be updated based on the interaction formulae, before going to the next iteration. The iteration procedure is summarized in Fig. 4.5.

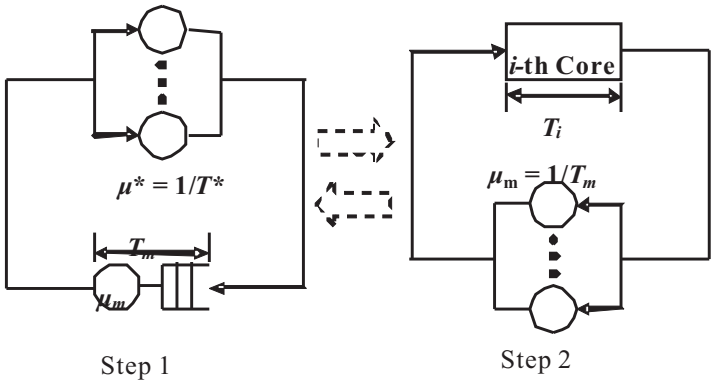

Figure 4.4. Iterative Procedure for Multicore Decoupling.

With this iterative procedure, an N_c-core system at the workload intensities $m_i i = 1 : N_c$ is decoupled into N_c single core systems. Single-core system i is composed of the ith core in the original system with the shared resource replaced by an $M/M/m_i$ queuing server local to the core. With all the workload intensities fixed for other cores, the utilization increment ratio for the ith core can then be evaluated by simulating the core at both workload intensities m_i and $m_i + 1$. As tested in [25], as long as $n \ll M$, adding the

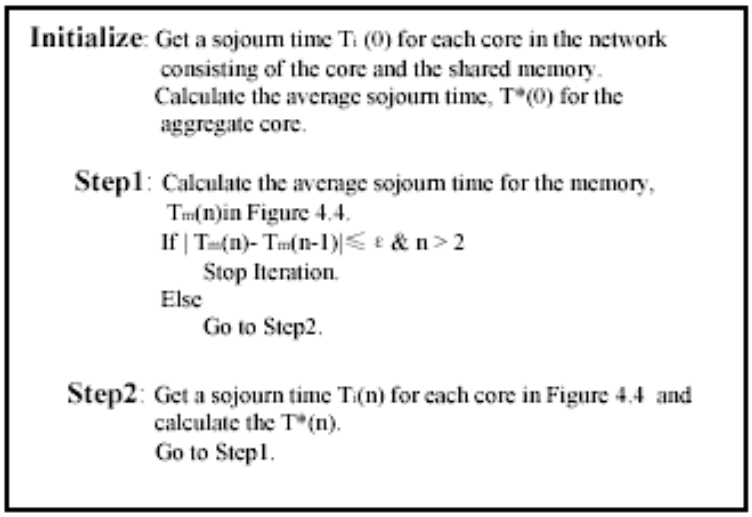

Figure 4.5. Iterative Algorithm.

workload intensity to the system by n will have little impact on the value of T_m , which is estimated at workload intensities $m_i i = 1 : N_c$ through the iterative procedure. This means that instead of having to reevaluate T_m for every unit of workload intensity added to the system, we only need to reevaluate it after adding n units of workload intensity to the system. To further reduce the computational complexity, instead of incrementing one unit at a time, we evaluate utilization increment ratios in the unit of n threads. In other words, we add n threads to a core with the minimal utilization increment ratio at a time. The second step of our heuristic can then be stated as follows:

1. Assign minimally required workload intensity m_i to each core;

2. Estimate T_m using the iterative procedure

3. With a user determined $n(\ll M)$, estimate utilization increment ratio for each core and increment the workload intensity by n for the core with the smallest utilization increment ratio; if the utilization of $M/M/m_i$ queuing server reaches one, exit, otherwise, go back to step 2.

This algorithm requires to run up to N_c single-core simulations in each iteration for the iteration procedure. The number of iteration is unknown. Our testing in [25] suggests that the number of iterations required is from a few to a few tens. After each iterative procedure is done, up to N_c single-core simulations need to be performed to finish step 3. This amounts to about a few tens of up to N_c single-core simulations. In practice, a many-core CMP has limited number of heterogeneous cores, e.g., less than ten. So in practice, we only need to perform up to a few hundred single-core simulations for adding n threads to the system. With our simulation tool [25], each single-core system simulation will take only a fraction of a second to finish on a Pentium III PC. This means that adding n threads to the system will take about a few minutes to finish. If each core can run up to 8 threads and initially each has 4 threads running, making the total initial workload intensity $M = 4000$ and there are up to 4000 additional threads to be added. Let $n = 4(\ll M)$. Then we need up to a few hours to complete step two. To program a CMP with 1000 cores and 8000 threads, spending a few hours on mapping the program-task-to-core and calculating the workload intensities in the initial programming phase appears to be reasonably short. Of course, step one of the heuristic requires the development of a parser that can analyze the pseudo code to generate the workload parameters for each core and then run step one, which adds more computational complexities.

4.5 Related Work

There are some existing fast algorithms for data path functions to CP core topology mapping (e.g., [41][42][43]). These algorithms, however, have 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 [42] accounts for certain multithreading effect, it only works for a single memory access and under a coarse-grained scheduling discipline.

CHAPTER 5

CONCLUSION AND FUTURE WORK

This dissertation proposed a novel chip multiprocessor (CMP) performance analysis methodology. In this methodology, a CMP is modeled generically at the thread level, overlooking the instruction-level and microarchitectural details. The aim is to allow various possible program-task-to-core mapping choices to be tested quickly in the initial programming phase, when the executable program is yet to be developed.

On the basis of this methodology, a analytical modeling technique based closed queuing network models and a framework for fast program-task-to-core mapping were developed. While the analytic modeling technique allows the general performance properties of a large design space to be characterized, the framework makes it possible to allow program-task-to-core mapping and workload intensity assignment to achieve maximal throughput/utilizations for many-core processors. All the approaches were allowing fast performance testing of CMPs with large numbers of cores and threads in the initial programming phase.

Our future work will be mainly concerned with the implementation and testing of the algorithms developed in Chapter 4.

APPENDIX A

CODE PATH

Table A.1. Code Path for *Generic IPv4 forwarding*(IXP1200)

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.2. Code Path for *ATM/Ethenet IP Forwading*(IXP1200)

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.3. Code Path for *Layer-2 Filtering*(IXP1200)

Task	# instructions in code segment ($t_{m,k} - t_{m-1,k}$)	I/O	$T_{j,k}$
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

APPENDIX B

SINGLE CORE MAIN RESULTS

Here we provide a detailed proof of the theorem given in Chapter 3.

Proof of Theorem: The theorem can be decomposed into two parts:

(1) if $\frac{m_0 a_i}{m_i} \leq 1, \forall i = 1, 2, \cdots, m_q$, then $\lim_{M_t \to \infty} P_I(0) = 0$.

(2) if there is at least one term in $\{m_0 a_i / m_i\}_{i=1:m_q}$ larger than 1, then $\lim_{M_t \to \infty} P_I(0) > 0$.

In what follows, we prove these two parts separately.

First, we prove the first part. Without loss of generality, assume $\frac{a_{m_q}}{m_{m_q}} = \max\left\{\frac{a_i}{m_i}\right\}$ and $\frac{a_1}{m_1} = \min\left\{\frac{a_i}{m_i}\right\}$, for $i \in [1, m_q]$. Dividing both the numerator and denominator of Eq. (3.17) by $\left(\frac{a_{m_q}}{m_{m_q}}\right)^{M_t}$, we have,

$$P_I(0) = \frac{\sum_{k_1+\cdots+k_{m_q}=M_t} \prod_{i=1}^{m_q} \left(\frac{a_i m_{m_q}}{m_i a_{m_q}}\right)^{k_i} \frac{m_i^{k_i}}{\beta_i(k_i)}}{\sum_{n=0}^{M_t} \frac{\left(\frac{a_{m_q}}{m_{m_q}}\right)^{n-M_t}}{\beta_0(M_t-n)} \sum_{k_1+\cdots+k_{m_q}=n} \prod_{i=1}^{m_q} \left(\frac{a_i m_{m_q}}{m_i a_{m_q}}\right)^{k_i} \frac{m_i^{k_i}}{\beta_i(k_i)}} \tag{B.1}$$

Since $\frac{a_1}{m_1} \leq \frac{a_i}{m_i} \leq \frac{a_{m_q}}{m_{m_q}}$, for $i = 2, \cdots, m_q - 1$, we have,

$$\left(\frac{a_1 m_{m_q}}{m_1 a_{m_q}}\right) \leq \left(\frac{a_i m_{m_q}}{m_i a_{m_q}}\right) \leq \left(\frac{a_{m_q} m_{m_q}}{m_{m_q} a_{m_q}}\right) = 1 \tag{B.2}$$

Hence,

$$P_I(0) \leq \frac{\sum_{k_1+\cdots+k_{m_q}=M_t} \prod_{i=1}^{m_q} \frac{m_i^{k_i}}{\beta_i(k_i)}}{\sum_{n=0}^{M_t} \frac{\left(\frac{a_{m_q}}{m_{m_q}}\right)^{n-M_T}}{\beta_0(M_t-n)} \sum_{k_1+\cdots+k_{m_q}=n} \prod_{i=1}^{m_q} \left(\frac{a_1 m_{m_q}}{m_1 a_{m_q}}\right)^{k_i} \frac{m_i^{k_i}}{\beta_i(k_i)}} \tag{B.3}$$

The queuing server q_i can be generally viewed as an M/M/m_i queue, $i \in (0, \cdots, m_q)$. Since for $\forall k_i$, $\beta_i(k_i) \geq m_i! m_i^{k_i - m_i}$ and $\beta_i(k) < m_i! m_i^{k_i}$ and noticing that there are $\frac{(M_t+m_q-1)!}{M_t!(m_q-1)!}$ elements in $\sum_{k_1+\cdots+k_{m_q}}$, we have:

$$
\begin{aligned}
P_I(0) &< \frac{\sum_{k_1+\cdots+k_{m_q}=M_t} \prod_{i=1}^{m_q} \frac{m_i^{m_i}}{m_i!}}{\sum_{n=0}^{M_t} \frac{\left(\frac{a_{m_q}}{m_{m_q}}\right)^{n-M_T}}{\beta_0(M_t-n)} \sum_{k_1+\cdots+k_{m_q}=n} \prod_{i=1}^{m_q} \left(\frac{a_1 m_{m_q}}{m_1 a_{m_q}}\right)^{k_i} \frac{1}{m_i!}} \\
&= \frac{\frac{(M_t+m_q-1)!}{M_t!(m_q-1)!} \prod_{i=1}^{m_q} \frac{m_i^{m_i}}{m_i!}}{\sum_{n=0}^{M_t} \left[\frac{\left(\frac{m_{m_q}}{a_{m_q}}\right)^{M_T-n}}{\beta_0(M_t-n)} \cdot \frac{(n+m_q-1)!}{n!(m_q-1)!} \cdot \left(\frac{a_1 m_{m_q}}{m_1 a_{m_q}}\right)^n \cdot \prod_{i=1}^{m_q} \frac{1}{m_i!} \right]} \\
&< \frac{1}{\left(\frac{M_{m_q}}{a_{m_q}}\right)^{M_t} \cdot \sum_{n=0}^{M_t} \frac{1}{\beta_0(M_t-n)} \left(\frac{a_1}{m_1}\right)^n \cdot \prod_{i=1}^{m_q} {m_i}^{m_i}} \\
&= \frac{1}{\left(\frac{M_{m_q}}{m_0 a_{m_q}}\right)^{M_t} \prod_{i=1}^{m_q} {m_i}^{m_i} \sum_{n=0}^{M_t} \frac{m_0^{M_t-n}}{\beta_0(M_t-n)} \left(\frac{a_1 m_0}{m_1}\right)^n} \\
&\leq \frac{1}{\left(\frac{M_{m_q}}{m_0 a_{m_q}}\right)^{M_t} \prod_{i=1}^{m_q} {m_i}^{m_i} \sum_{n=0}^{M_t} \frac{m_0^{M_t-n}}{m_0! m_0^{M_t-n}} \left(\frac{a_1 m_0}{m_1}\right)^n} \\
&= \frac{1}{\left(\frac{1}{m_0!} \prod_{i=1}^{m_q} {m_i}^{m_i}\right) \left(\frac{M_{m_q}}{m_0 a_{m_q}}\right)^{M_t} \cdot \frac{1-\left(\frac{a_1 m_0}{m_1}\right)^{M_t+1}}{1-\frac{a_1 m_0}{m_1}}}
\end{aligned}
\tag{B.4}
$$

Denote the last expression in Eq. (B.4) as $= P_I'(0)$. Since $\frac{a_1 m_0}{m_1} \leq \frac{a_{m_q} m_0}{m_{m_q}} \leq 1$, and $\lim_{M_t \to \infty} P_I'(0) = 0$. From Eq. (B.4), we have $P_I'(0) > P_I(0)$. Note that $P_I(0) \geq 0$. Hence $\lim_{M_t \to \infty} P_I(0) = 0$.

Now we prove the second part. To facilitate the proof, the dependency of $P_I(0)$ on m_q is explicitly included in $P_I(0)$ as a superscript, i.e., $P_I^{(m_q)}(0)$. Futhermore, since there is at least one term in $\{m_0 a_i / m_i\}_{i=1:m_q}$ larger than 1, we assume $\frac{m_0 a_1}{m_1} > 1$. For $m_q = 1$ (i.e., there is only one resource), from Eq. (3.17), we have

$$
P_I^{(1)}(0) = \frac{\frac{a_1^{M_t}}{\beta_1(M_t)}}{\sum_{k_1=0}^{M_t} \frac{a_1^{k_1}}{\beta_0(M_t-k_1)\beta_1(k_1)}} \tag{B.5}
$$

According to the assumption in part (2), $\frac{m_0 a_1}{m_1} > 1$; from Eq. (3.26) we have,

$$\lim_{M_t \to \infty} P_I^{(1)}(0) = \frac{1}{\frac{m_0^{m_0}}{m_0!} \cdot \frac{\left(\frac{a_1 m_0}{m_1}\right)^{-m_0+1}}{\frac{a_1 m_0}{m_1}-1} + \sum_{k_1=0}^{m_0-1} \frac{\left(\frac{a_1}{m_1}\right)^{-k_1}}{k_1!}} > 0 \tag{B.6}$$

This means that for the single resource case, the second part of the theorem holds true. To prove the theorem holds true in general, we need to show that it holds true for $\forall m_q$. Now if $P_I^{(m_q+1)}(0) \geq P_I^{(m_q)}(0)$ for $\forall m_q$, the second part of the theorem will hold true for $\forall m_q$. In the following, we show this is indeed the case.

For $\forall m_q$, Let

$$P_I^{(m_q)}(0) = \frac{\sum_{k_1+\cdots+k_{m_q}=M_t} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)}}{\sum_{n=0}^{M_t} \frac{1}{\beta_0(M_t-n)} \sum_{k_1+\cdots+k_{m_q}=n} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)}} \tag{B.7}$$

Notice that Eq. (B.7) is the same as Eq. (3.17). For $m_q + 1$, we have,

$$P_I^{(m_q+1)}(0) = \frac{\sum_{k_1+\cdots+k_{m_q}+k_{m_q+1}=M_t} \prod_{i=1}^{m_q+1} \frac{a_i^{k_i}}{\beta_i(k_i)}}{\sum_{n=0}^{M_t} \frac{1}{\beta_0(M_t-n)} \sum_{k_1+\cdots+k_{m_q}+k_{m_q+1}=n} \prod_{i=1}^{m_q+1} \frac{a_i^{k_i}}{\beta_i(k_i)}} \tag{B.8}$$

or

$$P_I^{(m_q+1)}(0) = \frac{\sum_{k_1+\cdots+k_{m_q}=M_t} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)} \quad + \quad \sum_{k_{m_q+1}=1}^{M_t} \frac{a_{m_q+1}^{k_{m_q+1}}}{\beta_{m_q+1}(k_{m_q+1})} \sum_{k_1+\cdots+k_{m_q}=M_t-k_{m_q+1}} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)}}{\sum_{n=0}^{M_t} \frac{1}{\beta_0(M_t-n)} \sum_{k_1+\cdots+k_{m_q}=n} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)} \quad + \quad \sum_{k_{m_q+1}=1}^{n} \frac{a_{m_q+1}^{k_{m_q+1}}}{\beta_{m_q+1}(k_{m_q+1})} \sum_{k_1+\cdots+k_{m_q}=n-k_{m_q+1}} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)}} \tag{B.9}$$

Eq. (B.9) is written in such a form that the first terms in both the numerator and the denominator are the same as the numerator and denominator in Eq. (B.8), respectively. Now, let L and R be the second term in the denominator multiplied by the first term in the numerator, and the first term in the denominator multiplied by the second term in the numerator, respectively, as given below:

$$L = \{\sum_{n=1}^{M_t} \frac{1}{\beta_0(M_t-n)} \cdot \sum_{k_{m_q+1}=1}^{n} \frac{a_{m_q+1}^{k_{m_q+1}}}{\beta_{m_q+1}(k_{m_q+1})} \cdot \sum_{k_1+\cdots+k_{m_q}=n-k_{m_q+1}} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)}\}$$
$$\cdot\{\sum_{k_1+\cdots+k_{m_q}=M_t} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)}\} \tag{B.10}$$

$$R = \{\sum_{n=0}^{M_t} \frac{1}{\beta_0(M_t-n)} \cdot \sum_{k_1+\cdots+k_{m_q}=n} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)}\} \cdot \{\sum_{k_{m_q+1}=1}^{M_t} \frac{a_{m_q+1}^{k_{m_q+1}}}{\beta_{m_q+1}(k_{m_q+1})}$$
$$\cdot \sum_{k_1+\cdots+k_{m_q}=M_t-k_{m_q+1}} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)}\} \tag{B.11}$$

Obviously, $P_I^{(m_q+1)}(0) \geq P_I^{(m_q)}(0)$ if and only if $R \geq L$. To show $R \geq L$, we construct another quantity L' as follows:

$$L' = \sum_{n=1}^{M_t} \frac{1}{\beta_0(M_t-n)} \cdot \sum_{k_{m_q+1}=1}^{n} \frac{a_{m_q+1}^{k_{m_q+1}}}{\beta_{m_q+1}(k_{m_q+1})} \cdot \sum_{k_1+\cdots+k_{m_q}=M_t-k_{m_q+1}} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)}$$
$$\cdot \sum_{k_1+\cdots+k_{m_q}=n} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)} \tag{B.12}$$

To prove $R \geq L$, we first show that $L' \geq L$ and then $R \geq L'$.

First, we note that the first two sums in Eq. (B.10) are the same as the first two sums in Eq. (B.12), except in Eq. (B.12), there is an extra term at $n = 0$. Clearly, if we could show that for any given $n > 0$, the last two sums in Eq. (B.12) is no less than the last two sums in Eq. (B.10), we have $L' \geq L$. In other words, we want to show $E' \geq E$, where,

$$E = \sum_{k_1+\cdots+k_{m_q}=n-k_{m_q+1}} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)} \cdot \sum_{k_1+\cdots+k_{m_q}=M_t} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)} \tag{B.13}$$

and

$$E' = \sum_{k_1+\cdots+k_{m_q}=M_t-k_{m_q+1}} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)} \cdot \sum_{k_1+\cdots+k_{m_q}=n} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)} \tag{B.14}$$

According to **Corrolary A** given in Appendix C, both E and E' have the same Z value, i.e., $Z = M_t + n - k_{m_q+1}$. In E, let A be the smaller one of M_t and $n - k_{m_q+1}$

and in E', A' be the smaller one of n and $M_t - k_{m_q+1}$. Note that if $A' \geq A$ then $E' \geq E$, since, according to **Corrolary A**, E and E' are monotonously increasing function when $A', A \in [1, \lfloor \frac{Z}{2} \rfloor]$. As a result, to prove $L' \geq L$, we need to show that $A' \geq A$.

Since $M_t \geq n \geq k_{m_q+1} \geq 1$, $M_t > n - k_{m_q+1}$, $A = n - k_{m_q+1}$. In E', since both n and $M_t - k_{m_q+1}$ can be smaller than $\lfloor \frac{Z}{2} \rfloor$, we have,

$$A' - A = \begin{cases} M_t - n & n > \lfloor \frac{Z}{2} \rfloor \\ k_{m_q+1} & n \leq \lfloor \frac{Z}{2} \rfloor \end{cases}$$

Again, since $M_t \geq n \geq k_{m_q+1} \geq 1$, $A' - A \geq 0$ always holds. So we have $E' \geq E$, and therefore, $L' \geq L$.

Finally, we show that $R \geq L'$. We first rewrite Eq. (B.12) as follows:

$$\begin{aligned} L' \;=\; & \sum_{n=1}^{M_t} \frac{1}{\beta_0(M_t - n)} \cdot \sum_{k_1+\cdots+k_{m_q}=n} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)} \cdot \sum_{k_{m_q+1}=1}^{n} \frac{a_{m_q+1}^{k_{m_q+1}}}{\beta_{m_q+1}(k_{m_q+1})} \\ & \cdot \sum_{k_1+\cdots+k_{m_q}=M_t-k_{m_q+1}} \prod_{i=1}^{m_q} \frac{a_i^{k_i}}{\beta_i(k_i)} \end{aligned} \tag{B.15}$$

We note that at any given n, the part with the last two sums in Eq. (B.11) is no less than the part with the last two sums in Eq. (B.15), because $M_t \geq n$ for $\forall n$. Furthermore, the parts involving the first two sums in Eq. (B.11) and Eq. (B.15) are the same, except that the part in Eq. (B.11) has an extra term at $n = 0$. Hence, we have $R \geq L'$. Since we have shown that $L' \geq L$, $R \geq L$ and therefore $P_I^{(m_q+1)}(0) \geq P_I^{(m_q)}(0) \geq P_I^{(1)}(0) > 0$.

APPENDIX C

PROOF OF COROLLARY A

First, we prove the following theorem.

Theorem A: There are N boxes ($N \geq 2$). The capacity of the ith box ($i = 1, 2, \cdots, N$) is k_i, and $\sum_{i=1}^{N} k_i = M$. $S_N(y)$ is the number of different ways to put y identical balls into these N boxes. Then $S_N(y)$ is a monotonously increasing function of y, when $y \in [1, \lfloor \frac{M}{2} \rfloor]$, and $S_N(y)$ reaches its maximal value when $y = \lfloor \frac{M}{2} \rfloor$.

Proof: We prove it by induction.

Basis step ($N = 2$): we define a step function $u(x)$ as follows:

$$u(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{C.1}$$

$S_2(y)$ can be calculated as follows: first, assume that there is no capacity constraints for both boxes (i.e. $k_1 = \infty, k_2 = \infty$). Then there are $\frac{(y+1)!}{y! \cdot 1!}$ different ways to put y balls into these two boxes. However, since the size of the boxes is not infinity, we can only put at most k_1 balls in box 1, so the number of ways by which we can put more than k_1 balls in the 1st box must be excluded, which is $\sum_{j_1=1}^{y} u(j_1 - k_1)$. Similarly, for the 2nd box, there are $\sum_{j_2=1}^{y} u(j_2 - k_2)$ number of different ways needs to be excluded. Therefore, for $y, y+1 \leq \lfloor \frac{M}{2} \rfloor$, we have,

$$S_2(y) = \frac{(y+1)!}{y! \cdot 1!} - \sum_{j_1=1}^{y} u(j_1 - k_1) - \sum_{j_2=1}^{y} u(j_2 - k_2) \tag{C.2}$$

$$\begin{aligned} S_2(y+1) &= \frac{(y+2)!}{(y+1)! \cdot 1!} - \sum_{j_1=1}^{y+1} u(j_1 - k_1) - \sum_{j_2=1}^{y+1} u(j_2 - k_2) \\ &= \frac{(y+2)!}{(y+1)!} - \sum_{j_1=1}^{y} u(j_1 - k_1) - \sum_{j_2=1}^{y} u(j_2 - k_2) - u(y+1-k_1) \\ &\quad -u(y+1-k_2) \end{aligned} \tag{C.3}$$

We further have,

$$\begin{aligned} S_2(y+1) - S_2(y) &= \frac{1}{y+1} \cdot \frac{(y+1)!}{y!} - [u(y+1-k_1) + u(y+1-k_2)] \\ &= 1 - [u(y+1-k_1) + u(y+1-k_2)] \end{aligned} \tag{C.4}$$

We want to show $S_2(y+1) - S_2(y) \geq 0$ or equivalently, $[u(y+1-k_1) + u(y+1-k_2)] \leq 1$.

For $u(y+1-k_1)$

If $(y+1-k_1) \leq 0$ then $u(y+1-k_1) = 0$, $[u(y+1-k_1)+u(y+1-k_2)] = u(y+1-k_2) \leq 1$, and $S_2(y+1) - S_2(y) \geq 0$.

else if $(y+1-k_1) > 0$ then $u(y+1-k_1) = 1$ and $y+1 > k_1$

$u(y+1-k_2) = u(y+1-M+k_1)$, because $k_2 = M - k_1$.

$\because (y+1-k_1) > 0$,

$\therefore y+1 > k_1$. $\because y+1 \leq \lfloor \frac{M}{2} \rfloor$,

$\therefore k_1 < y+1 \leq \lfloor \frac{M}{2} \rfloor$

$\therefore (y+1-M+k_1) < 0, \therefore u(y+1-k_2) = u(y+1-M+k_1) = 0$ and $[u(y+1-k_1) + u(y+1-k_2)] \leq 1$.

$\therefore S_2(y+1) - S_2(y) \geq 0$, i.e., the theorem holds true for $N = 2$.

Induction hypothesis ($N \geq 2$): $S_N(y+1) \geq S_N(y)$, for $y, y+1 \leq \lfloor \frac{M}{2} \rfloor$

Induction step: now consider $N+1$ boxes and y balls ($y, y+1 \leq \lfloor \frac{M}{2} \rfloor$). Here we consider the first N boxes as one group and the $(N+1)$th box as the other group. Consider putting j balls into the $(N+1)$th box, and the rest $(y-j)$ balls into the first N boxes. The different ways to put $(y-j)$ balls into the first N boxes is given by $S_N(y-j)$. We may put j ($j \in [0, k_{N+1}]$) balls into the $(N+1)$th box. We have,

$$S_{N+1}(y) = \sum_{j=0}^{y} u(k_{N+1} - j) S_N(y-j) \tag{C.5}$$

Similarly, we have,

$$S_{N+1}(y+1) = \sum_{j=0}^{y+1} u(k_{N+1} - j) S_N(y+1-j) \tag{C.6}$$

Every term in Eq. (C.6), i.e., $u(k_{N+1}-j)S_N(y+1-j)$, is no less than the term in Eq. (C.5), since $S_N(y+1-j) \geq S_N(y-j)$ (induction hypothesis). Furthermore, Eq. (C.6) has one more term $u(k_{N+1}-(y+1))S_N(y+1-(y+1))$, which is non-negtive. Therefore, $S_{N+1}(y+1) \geq S_{N+1}(y)$, for $y, y+1 \leq \lfloor \frac{M}{2} \rfloor$

Corollary A: For polynomial function $F = \sum_{k_1+\cdots+k_n=A} \prod_{i=1}^{n} a_i^{k_i^1}$ $\sum_{k_1+\cdots+k_n=B} \prod_{i=1}^{n} a_i^{k_i^2}$, and $A+B=Z$, Z is a constant. Assume $A \leq B$. Then F reaches its maximal value when $A = \lfloor \frac{Z}{2} \rfloor$, and F is a monotonously increasing function of A, for $A \in [1, \lfloor \frac{Z}{2} \rfloor]$

Proof: We define:

$$\begin{aligned} F &= \sum_{k_1+\cdots+k_n=A} \prod_{i=1}^{n} a_i^{k_i^1} \sum_{k_1+\cdots+k_n=B} \prod_{i=1}^{n} a_i^{k_i^2} \\ &= \sum_{k_1^0+\cdots+k_n^0=Z} C(k_1^0, \cdots, k_n^0) \cdot \prod_{i=1}^{n} a_i^{k_i^0} \end{aligned} \tag{C.7}$$

We have $A+B=Z$, and $k_i^1 + k_i^2 = k_i^0$. $C(k_1^0, \cdots, k_n^0)$ (denoted by C for convenience) is the coefficient for each term in Eq. (C.7). For given A and B, the value of F is determined by C. The question of how to calculate C can be mapped to a combinatorial problem below:

Suppose we have n boxes. The capacity of the ith box is k_i^0 ($\sum_{i=1}^{n} k_i^0 = Z$). It turns out that C is the number of ways to put A identical balls into these boxes. This combinatorial problem is addressed in **Theorem A**. Since $A+B=Z$, and Z is a constant, we just have one variable. For convenience, we assume that A is the smaller than B. According to **Theorem A**, C reaches its maximal value when $A = \lfloor \frac{Z}{2} \rfloor$, and C is a monotonously function of A, for $A \in [1, \lfloor \frac{Z}{2} \rfloor]$. Therefore, F reaches its maximal value when $A = \lfloor \frac{Z}{2} \rfloor$, and F is a monotonously function of A, for $A \in [1, \lfloor \frac{Z}{2} \rfloor]$.

REFERENCES

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

[2] H. Che, C. Kumar, and B. Menasinahal, " A Fast Latency Bound Estimation Algorithm for a Multithreaded Network Processor", *the 18th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, Nov. 2006.

[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] 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, 248-260, Apr. 1975.

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

[6] " Teletraffic Engineering ", ITU-D Study Group 2, 2002.

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

[8] C. K. Chow. Determination of Cache's Capacity and its Matching Storage Hierarchy, *IEEE Transactions on Computers*, c-25, 157 - 164, 1976. 811 - 825, 1992.

[9] H. Jung, Ph.d Dissertation.

[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)*.

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

[12] 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.

[13] 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.

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

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

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

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

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

[23] Y. N. Lin, Y. D. Lin, Y. C. Lai, “Thread Allocation in Chip Multiprocessor Based Multithreaded Network Processors”, in *Proc. of the 22nd International Conference on Advanced Information Networking and Applications (AINA 2008)*, pp.718-725, Mar. 2008.

[24] S. Subramaniam, M. Prvulovic, and G. H. Loh, “PEEP: Exploiting Predictability of Memory Dependences in SMT Processors”, in *Proc. of the 14th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2008.

[25] H. Jung, M. Ju, H. Che, “A Theoretical Framework for Design Space Exploration of Manycore Processors”,In the Proceeding of *MASCOTS 2011*, July, 2011.

[26] M. Ju, H. Jung, H. Che, “A Performance Analysis Methodology for Multi-core, Multithreaded Processors”, submitted to *IEEE Transactions on Computers*.

[27] M. Ju, H. Jung, H. Che, “Fast Performance Analysis of Many-core Processors”, to appear as an invited chapter in the book on “Scalable Computing and Communications: Theory and Practice” editted by S. Khan, L. Wang, and A. Zomaya, John Wiley & Sons, 2011.

[28] J. Anselmi, P. Cremonesi, “A unified framework for the bottleneck analysis of multiclass queuing networks”, *Performance Evaluation*,Vol. 77, No. 4, pp. 218-234, April, 2010.

[29] P. Denning, J. Buzen, “The Operational Analysis of Queueing Network Models”,*Journal of ACM Computing Surveys (CSUR)*, Vol. 10, No. 3, pp. 225-261 September, 1978.

[30] G. Bolch, S. Greiner, H. Meer, K. S. Trivedi, *Queuing Networks and Markov Chains, 2nd Edition*, A John Wiley & SONS, Inc., 2006.

[31] 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

[32] 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

[33] 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

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

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

[36] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. “A Predictive Model for Superscalar Processor Performance”, in Proc. of *Intl. Symp. on Microarchitecture*, pp. 161-170, Dec. 2006.

[37] B. C. Lee and D. M. Brooks. “Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction”, in Proc. of *Intl. Conf. on Archi-*

tectural Support for Programming Languages and Operating Systems, pp 185-194, Oct. 2006.

[38] C. Hughes and T. Li, "Accelerating Multi-Core Processor Design Space Evaluation Using Automatic Multi-Threaded Workload Synthesis", in Proc. of *IEEE Int'l Symp. Workload Characterization (IISWC)*, pp. 163-172, Sept. 2008.

[39] S. Nussbaum and J.E. Smith, "Statistical Simulation of Symmetric Multiprocessor Systems", in Proc. of *35th Ann. Simulation Symp.*, pp. 89-97, Apr. 2002.

[40] 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

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

[42] 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.

[43] 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.

BIOGRAPHICAL STATEMENT

Miao Ju was born in China, in 1983. He received his B.S. degree from BeiHang University, Beijing, China, in 2005, his M.S. degrees from The University of Texas at Arlington in 2007 in Computer Science and Engineering. His current research interest is in the area of parallel architecture and processing in computer system. He is a member of IEEE society.