Performance Modeling and Analysis of Multithreaded Architectures

Shashank S. Nemawarkar

McGill University, Montreal

Department of Electrical Engineering

August 1996

A Thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ISBN 0-612-19756-5

Dedicated to my constant source of inspiration, my parents

Sharadchandra and Sheelprabha Nemawarkar

Abstract

Multithreaded architectures use the parallelism in programs to tolerate long latencies for communications and synchronizations. On encountering a long latency memory access, the processor in a multithreaded system rapidly switches context to another computation thread, thereby improving the performance. Unlike traditional single threaded execution and multitasking in operating systems, multithreading allows accesses from one or more threads of a user program at a processor to contend for system resources simultaneously. Hence, a performance analysis of multithreading should account for the effect of multiple concurrent accesses on throughput of subsystems.

Modeling a real multithreaded system, like McGill's EARTH system, poses several problems. First, in realistic subsystem interactions, more than one subsystem may serve the same access simultaneously, so contentions are difficult to predict. Second, the thread characteristics like the number of remote accesses can differ with processing nodes. Thus, an accurate computation of delays at subsystems is essential.

We propose analytical performance models, develop solution techniques, validate model predictions, and analyze the performance of multithreaded architectures. Our analytical models, based on closed queueing networks, account for the feedback effect of the load at subsystems on the processor performance. We demonstrate the robustness of these closed queueing network models over open system models for the performance prediction. With the feedback effect and the iterative nature of our solution technique, we predict the performance of complex subsystem interactions in the EARTH system under a multithreaded workload. Measurements from actual program executions are within 5% to 20% of model predictions.

The model inputs are the architectural parameters and program workload characteris-

tics. Model predictions include the processor utilization, message rate to the network, and latency for remote accesses.

Given a program workload, we show the effectiveness of multithreading to *tolerate* communication latencies. We show the significance of the *network capacity* to tune program workload characteristics to achieve high performance. Our analysis of the EARTH system shows that under a multithreaded program workload, subsystem interactions at processing nodes are the bottlenecks. Reducing access times for subsystems in an EARTH node leads to a performance improvement especially at fine thread granularities. Multithreading provides more robust performance to the changes in data distributions than a single threaded execution. Our results demonstrate the tradeoffs of realistic costs of multithreading on the performance of fine-grain parallel program workload.

Overall, our analytical models are useful to system architects and compiler writers to provide insight to the performance related optimizations.

Résumé

Les architectures multiprogrammées prennent en compte le parallélisme des programmes afin de tolérer de plus longues latences dans les communications et les synchronisations. Lors d'un accès mémoire avec une longue latence, le processeur d'un système multiprogrammé commute rapidement de contexte vers un autre programme de calcul, ce qui améliore la performance. Contrairement à l'exécution traditionnelle monoprogrammée ainsi que dans les systèmes d'exploitation multitâches, la multiprogrammation permet des accès concurrents et de façon simultannée aux resources du système global, à partir d'un ou de plusieurs threads d'un même programme usager. Une analyse de performance en multiprogrammation doit prendre en compte l'effet des accès concurrents et multiples sur le débit des sous-systèmes.

La modélisation d'un système réel multiprogrammé, comme le système EARTH de McGill, pose d'autres problèmes. Premièrement, lors des interactions réelles entre sous-systèmes, plus d'un sous-système pourrait émettre le même accès au même moment, ce qui rend la prédiction des conflits difficile. Deuxièment, les caractéristiques des threads tel que le nombre d'accès déportés peut changer selon les noeuds de traitement. Par conséquent, un calcul exact des délais s'avère essentiel.

Nous proposons, dans cette thèse, des modèles de performance analytiques, nous développons des techniques de résolution, nous validons les prédictions du modèle et enfin nous analysons la performance des architectures multiprogrammées. Nos models analytiques, qui sont basés sur des réseaux de file d'attente fermés, considèrent l'effet rétroactif de la charge de traitement des sous-systèmes sur la performance du processeur. Nous démontrons la robustesse de ces modèles de réseaux de file d'attente fermés par rapport aux models ouverts pour la prédiction de la performance. Etant donné l'effet rétroactif et la nature itérative de

notre technique de résolution, nous prédisons la performance des interactions complexes entre sous-systèmes dans le système EARTH pour une charge de traitement multiprogrammée donnée. Les mesures prélevées à partir des exécutions actuelles de programmes se situent entre 5% et 20% des valeurs prédites par le modèle.

Les entrées du modèle sont les paramètres de l'architecture et la caractéristique de la charge de traitement du programme. Les prédictions ou sorties du modèle incluent l'utilisation du processeur, le taux de messages transférés sur le réseau ainsi que la latence des accès déportés.

Pour une charge de traitement donnée, nous montrons l'efficacité de la technique de multiprogrammation à supporter les latences de communication. Nous montrons aussi l'importance de la capacité du réseau à s'adapter aux caractéristiques de la charge de traitement du programme afin d'atteindre une performance élevée. Notre analyse sur le système EARTH montre que pour une charge de traitement donnée d'un programme multiprogrammé, les interactions entre les sous-systèmes, au niveau des noeuds de traitement, constituent les goulots d'étranglement. Réduire le temps d'accès des sous-systèmes dans un noeud du système EARTH permet d'améliorer la performance surtout à des niveaux fins de la granularité des programmes. La multiprogrammation permet d'atteindre de meilleures performances, en présence des changements dans la distribution des données, par rapport à une exécution monoprogrammeé. Nos résultats montrent les compromis possibles, avec des coûts réels d'une multiprogrammation, sur la performance des programmes parallèles à grain fin.

Enfin, nos models analytiques sont d'une grande utilité aux concepteurs d'architecture de systèmes et aux concepteurs de compilateurs en leur fournissant des indices sur les optimisations reliées à la performance.

Acknowledgements

During the Ph.D. program, I have benefited from direct and indirect interactions with many people. I take this opportunity to thank them.

I am very thankful to the constant guidance and support by my advisor, Prof. Gao. This thesis as well as my research outlook have tremendously benefited from his knowledge, enthusiasm, inquisitiveness, and ability to see beyond the written word.

Prof. V.K. Agarwal has been instrumental in providing me this opportunity for Ph.D. I would like to thank him for his guidance in the early part of this research, and the financial support through research grants.

During Prof. Govindarajan's stay at McGill University, I had very fruitful discussions on my research work. He was almost my unofficial co-supervisor. This collaboration laid the foundation of this thesis work and several publications [65, 66, 67].

The early ideas on the latency tolerance developed through my discussions with Prof. Bhatt. I deeply appreciate his support on technical and other aspects.

My thanks are also due to Ph.D. supervisory committee members, Prof. Tropper, Prof. Rajski, and Prof. Szymanski. The committee's insistence on the validation using a real system increased the contributions of this thesis significantly.

Prof. Anant Agarwal provided an opportunity to visit his Alewife group at MIT. His own technical contributions to the field of multithreaded architectures have also greatly influenced my thesis work.

A useful interaction with Prof. Andrew A. Chien resulted in a significant work reported in Section 5.8.2.

A collaboration with Dr. Xinmin Tian developed into a significant research, which is reported in Section 8.5.

This work could not have been in its present shape without an excellent, novel testbed to validate my ideas. I thank the EARTH Group at McGill University and the MANNA Group at GMD in Germany, for building the EARTH-MANNA multithreaded multiprocessor system. In particular, I would like acknowledge Dr. Olivier Maquelin for his technical support on the run-time system.

Ashay has been a great help in proof-reading the thesis. His support on almost every non-technical aspects has been very valuable.

I greatly appreciate Nilanjan and Sanjeev's help in proof-reading of this thesis, and Houria's help in translating the abstract into French. Sreedhar, Sandeep, Nagesh and Palash have proof-read several of my papers.

The MACS staff provided an excellent technical and secretarial support throughout my Ph.D. duration. Jacek has been a tremendous help on system related problems. Lynn, Trang and Christine were excellent during their stays with the MACS Lab. The system administration staff in the ACAPS laboratory was also great. Students in both the labs made the place enjoyable.

I acknowledge the financial support by MICRONET, Network Centers of Excellence, Canada. I would like to thank the administrative help from the staff at the Department of Electrical Engineering and the School of Computer Science at McGill University.

I thank Bala for friendship, encouragement, and a nice time during a significant part of my Montreal stay. Sreedhar and Srini were a great help in the difficult times.

My constant source of inspiration has been the dedication and hard work of my parents, Sharadchandra Nemawarkar and Sheelprabha Nemawarkar. Their tremendous influence on my life has been phenomenal.

My wife, Anjali, has been through thick and thin of my student life. This thesis is a result of her sacrifice of her own career aspirations while being supportive to me especially through the trying times.

Claim of Originality

This thesis contributes to the following aspects of multithreaded architectures: *analytical performance modeling*; *performance analysis*; and *performance optimizations.*

Analytical performance modeling:

- We propose analytical models to predict the performance of multithreaded architectures. We model them as integrated systems, i.e. with processor, memory and network subsystems, using closed queuing networks. We show the robustness of closed queueing networks for performance prediction over open system models.
- To extend our performance model to McGill's EARTH multithreaded system, we develop heuristics to the mean value analysis (MVA). First, we model the simultaneous possession of the bus at a processing node, when the memory or network interface is accessed. We exploit the iterative nature of the MVA, and the feedback effect of a closed system model, to obtain the solution using only one queuing network model. Second, we model a realistic multithreaded workload, i.e. thread characteristics at different processing nodes may differ. To compute the queueing delays accurately, our heuristic uses the service demand for each access in the queue at a subsystem.

Performance analysis:

- We show, under a multithreaded program execution model, how to derive the performance measures like processor utilization, message rate to the network, and latency for remote accesses with split-phase operations. We analyze the variation in these performance measures with architectural and workload parameters. Simulation results from the petri net models and measurements from program executions on the EARTH system validate model predictions.

- We propose a metric, *tolerance index*, to quantify the effectiveness of multithreading to tolerate latencies at a subsystem. The tolerance index, say for the network latency, shows how close is the system performance to that of an *ideal* system which incurs no network delays.

- We show the feedback effect about how significant are the throughputs of subsystems in tuning the multithreaded program workload characteristics. First, we show that the thread runlengths larger than the value of *effective memory latency* yield a high performance at a processing node. The effective memory latency is the average time between successive memory responses. Second, we show that the processor utilization increases with an increasing number of threads, as long as the message rate is not close to the *network capacity*. The network capacity is the maximum message rate per processor delivered by the network under a given access pattern. The increase in the processor performance is in spite of increases in network latencies and message rate to the network.

- Multithreaded operations on the EARTH system are composed of a sequence of simple operations such as memory accesses and network messages. We provide the first detailed characterization of a multithreaded program workload with real costs (on the EARTH system) and numbers of multithreading operations and local accesses.

Performance optimizations:

- Through model predictions and empirical measurements on the EARTH system, we demonstrate the tradeoffs of realistic costs of multithreading on the performance for fine-grain parallel program workloads. We also show that the performance of a multithreaded program execution is more robust to the changes in data distributions than a single threaded execution.

- We show that performance bottlenecks on the EARTH system are the subsystem interactions at processing nodes. By reducing access times of these subsystems, the performance improves at finer thread granularities. Such architectural configurations can be studied through our analytical models.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Multiprocessor architectures present an attractive approach to a cost-effective high performance computing. Multiprocessor systems consist of processing nodes on an interconnection network. A typical processing node is composed of, a processor, a local memory, and a network interface. On current distributed shared memory systems, costs of accessing data from a remote memory are an order of magnitude higher than the cost of accessing from the local memory [57, 70, 5, 46]. A good decomposition of the computation in a parallel program and a distribution of the data on a machine avoids excessive message traffic on the network. For such programs, a multiprocessor system yields a good performance with increase in the number of processors. When the sharing of data among processors increases, the network messages experience longer latencies due to an increased traffic [99]. Long latencies for communication across the network and synchronization in parallel program executions, are considered to be the important causes for the performance degradation of multiprocessor systems [14].

Multithreaded architectures are proposed as a promising approach to tolerate long communication latencies and unpredictable synchronization delays in parallel program executions. Examples of such systems include, TERA [9], MASA [41], Alewife [5], *T[68], TAM [26], RWC-1 [82], EARTH [46], Cilk [16], and M-Multicomputer [37]. Multithreading technique provides a split-phase mechanism for long latency operations, and a mechanism to rapidly switch the context of a computation task (thread). When a long latency operation occurs, the multithreading technique rapidly switches the context to another computation

task. This overlap of communication on one thread with computation on the other reduces the idle time at a processor. Thus, the processor performance improves by maintaining a pool of ready threads for execution, as well as by employing split-phase transactions for remote accesses and synchronizations. Next, we describe an abstraction of the multithreaded program execution. Later in Chapter 2, we discuss a multithreaded program workload in detail.

1.1 A Multithreaded Program Execution Model

The execution at a single-threaded processor progresses along the instructions fetched by an *instruction pointer* and a *stack pointer*. Together, these pointers are referred as an *activity specifier* in the literature [33]. Along with the registers, an activity specifier represents the *context* of a computation task (thread). The state of a single-threaded machine is made up of the *context* and the set of values held in the memory.

In a multithreaded program execution, multiple contexts co-exist at a processor. A context and associated set of values in the memory corresponds to the state of one computation thread. The computation progresses according to the activity specifiers for these threads. A multithreaded program workload is a partial order of multiple threads of computation. A *thread* is a sequence of instructions followed by split-phase long latency transactions, e.g., a remote memory access. There are two multithreading approaches to support the maintenance of context for a thread. The first is the *dataflow* style multithreading, in which each thread is atomic. That is, once scheduled the thread executes till its completion and stores its result in memory. Memory locations are used to communicate variables between threads. Registers lose their identity on the completion of a thread, and are not saved. The completion of a thread triggers synchronization among threads and further computation. This producer-consumer synchronization is an abstract machine model of multithreading with data-driven semantics. In an implementation such as the EARTH system [46], a compiler and runtime system may retain some registers across thread boundaries for an efficient execution, while not violating the abstract model. The second is the *von Neumann* style multithreading, in which a context (including registers) is retained across split-phase long latency transactions. When long latency split-phase memory accesses are issued, the thread is suspended and the state is saved at some place in the memory or in a register

bank. On completion of a long latency operation, an execution on a thread may continue after restoring the state of the thread in registers [9, 100, 5, 90].

In this thesis, we assume atomic threads as in the *dataflow* style of multithreading. We note that the *von Neumann* style multithreading can be captured by this program execution model, as testified by our experiments in Chapter 7.

1.2 Performance Issues of Multithreading

The composition of a split-phase transaction is crucial to its performance. In this thesis, split-phase transactions are also referred as multithreading operations. A multithreading operation is composed of tasks such as, receiving messages from the network, accessing the local memory, responding to messages, and performing some synchronization operations. Costs of multithreading operations include, an overhead for switching the context to the execution on another thread, and a support for split-phase accesses. Multithreading operations on multiple threads may lead to multiple outstanding requests in the system. Increased contentions at the memory and interconnection network may further increase the latencies. Performance of a multithreaded architecture depends on interactions of key components of a system under overlapped multithreaded computation and communication.

An architect attempts to alleviate bottlenecks to the performance on target applications. To efficiently support multithreading operations, an architect should know the following: *How frequently does a multithreading operation occur?* And, *which functional units have large response times?* There are two possible approaches to reduce the response time of a particular functional unit. One approach changes the organization of a subsystem e.g. memory, while other changes the implementation of a multithreading operation, e.g., how often does a processor check messages at the network interface. Thus, an architect needs to assess the effect of these changes on the performance.

One of the main objectives of a compiler (and a programmer) is to maximize the processor utilization on a given multiprocessor system for an application program. Performance related optimizations by a compiler change two aspects of a program workload: the *data distribution* and the *computation decomposition* [10, 69]. A compiler needs information on the following three aspects. First, *which characteristics of a multithreaded program work-*

load are significant to achieve high performance? Some program workload characteristics are, the number of threads, their runlengths, and the remote memory access pattern. A compiler should be able to vary these characteristics to achieve a desired performance. Second, *how do the variations in these program workload characteristics affect the performance of a system? In isolation as well as in combination?* A characterization under realistic costs for multithreading operations helps a compiler to choose suitable performance optimizations. Third, *what are the ranges of program workload characteristics which yield high performance?* Targets of performance optimizations are these ranges of workload characteristics.

Differences in performance optimization strategies for multithreaded systems with those for single threaded multiprocessor systems are as follows. For a single threaded system, the optimizations aim to reduce the network latencies, through a careful data partitioning, a reduced data sharing and a reduced network traffic [34, 99]. In contrast, one focus of the data distribution and computation decomposition on multithreaded systems is to remove the unnecessary serialization in computations. For example, a distribution of adjacent rows of an array to different memory modules reduces the access contentions to the same memory module. Thus, a sufficient number of threads is unraveled so that the computation on them is overlapped with the communication necessary for their progress. Parallel threads improve the processor utilization, however an increased number of split-phase network accesses increases the network latencies as a side-effect.

The focus of performance optimization strategies for traditional multitasking systems differ from multithreaded systems as follows. A multitasking operating system uses multiple tasks to improve the throughput of a computer system. Each task is assigned a specific time-slice for execution on the processor(s). No contentions from resources occurs among different tasks during a time-slice, hence the latencies at subsystems are low. In contrast, the multithreading technique allows a tightly coupled sharing of data among multiple threads on the same application. That is, these threads may co-exist at a processor, and share data at fine granularities such as every tens or hundreds of instructions [85, 77, 64]. The execution time of an application reduces with improved processor utilization. Accesses from multiple threads on the same processor, however, are allowed to contend for the system resources. So, the latencies to access these resources increase.

Thus, an increased overlap of computation and communication among multiple threads

requires changes to the traditional performance related optimizations.

1.3 Objectives and Research Issues

The objectives of this thesis are motivated by the needs of users[1] (architects and compilers) of multithreaded systems. Section 1.2 outlines the need to model an overlapped computation and communication in multithreaded program execution and to demonstrate the use of these models for performance optimizations. Thus, the main objectives of this thesis are two-fold:

- *To predict the performance of multithreaded architectures.* This objective includes the following aspects:
  1. *identify* the significant program workload parameters and architectural interactions which affect the performance;
  2. *develop* analytical models to predict the performance of multithreaded architectures; and
  3. *validate* model predictions using the results from simulations as well as system measurements.
- *To apply the analytical models for the performance analysis and optimizations of multithreaded architectures.* This objective includes the following aspects:
  1. *characterize* the performance of multithreaded architectures with changes in architectural and program workload parameters;
  2. *identify* the performance bottlenecks in program executions; and
  3. *demonstrate* how optimizations of program workload characteristics and architectural implementations can improve the performance of multithreaded architectures.

We consider above objectives within the context of multithreaded architectures and obtain solutions for each of the above steps. We also show the changes in the complexity

[1]Henceforth, we will use the term *users* to refer to *architects*, *compilers* and *programmers*, collectively. We will also refer *compilers* in place of *compilers* and *programmers*.

and scope of these problems with changes in the underlying system– a single processor system, a multiprocessor system or a real machine.

1.4 Overview of The Thesis

Our approach to meet the above mentioned objectives is as follows. We formulate performance models of multithreaded architectures using closed queueing networks. Our solution technique is based on approximate mean value analysis (MVA) [56]. We develop heuristics to account for multithreaded workload, and subsystem interactions like simultaneous resource possession. Results of Stochastic Timed Petri Net simulations verify the accuracy of our model predictions. Measurements from actual program executions on the EARTH system validate the model predictions. We extensively characterize the variation in performance measures using model parameters. Using the realistic costs for multithreading operations, we analyze the performance of the EARTH system. Through examples, we demonstrate the usefulness of our performance analysis to optimize the performance on multithreaded systems.

1.4.1 Performance Modeling

We model multithreaded architectures as integrated systems. The models account for the behavior of processors, memories and interconnection networks, and interactions among them under various program workloads. We show the following advantages of using closed system models like ours. *First,* by accounting for the feedback effect of the load of the subsystems on the processor performance, the model predictions are robust even when the system operates near the network saturation. We show tradeoffs of three open system models employing feedback to improve accuracy in their processor performance predictions with respect to the closed system models. *Second,* input parameters to our models can be directly supplied by the users. In contrast, open system models require input parameters like the message rate to the network, which are not usually known *a priori* (such models have been used in [80, 8, 18, 90]). *Finally,* our closed system model provides a broader picture of the system performance. For example, an increased locality in remote access pattern due to a different data distribution, can increase the message rate while simultaneously reducing

the network latency. If an open system model is used for the performance prediction [30, 1], the user needs to accurately estimate the message rate (since the network latency rises sharply with the message rate) and use a distinct characterization of network latency for different locality patterns. Thus, to achieve the same desired results, a user of open system models requires not only the statically known input parameters but also the intermediate output parameters.

Our choice of closed queueing networks to model multithreaded architectures is based on the following reasons. *First*, systems with a large number of processors can be analyzed quickly using standard techniques like the mean value analysis (MVA). Our model takes less than one minute to analyze a 64-processor system on SPARCStation-20. *Second*, models of a modified system can be developed quickly. As a case study, we have adapted our analytical model, which is originally developed for a multithreaded system with a 2-dimensional mesh network, to predict the performance of the EARTH system with an interconnection based on crossbar switches. *Third*, the MVA is amenable to heuristics. For the EARTH system model, we developed heuristics for realistic subsystem interactions. *Fourth*, these queueing network models have been applied to real systems in practice [58].

We applied our analytical model to the performance predictions of McGill's EARTH system. We proposed two approximations to the MVA. The first approximation models the realistic interactions at processing nodes in the EARTH system. The second approximation characterizes a realistic multithreaded workload.

First, on the architectural aspect, we model the *simultaneous possession* of the bus for accesses in an EARTH node. When an access from the processor is serviced by the local memory, the bus at the processing node is busy until the access completes. For this duration, no other resources at the node can be accessed, e.g. an access to the network interface from a synchronization unit does not go through. Previous solutions to a general problem of *simultaneous resource possession* involved the use of multiple queueing network models, which are solved iteratively. For example, one model for each processing node to account for processor accessing the local memory, and another model for the subsystems excluding processors at each node. These solutions are reported by Jacobson and Lazowska [48], Lazowska *et al.* [56], and de Souza e Silva and Muntz [31]. In context of the EARTH system, we exploit the iterative nature of the MVA to formulate the above problem under one analytical model of the complete system. Our heuristic is that the total queueing delay

for a new access to any resource on the bus is a sum of the service time for each already queued access through the bus, rather than the queueing delay at an individual resource alone.

Second, on the program workload aspect, the thread characteristics at different processing nodes may differ. Each type of request (e.g. local or remote memory accesses) requires a different service time from the server (memory system). So, a single unified queue length alone, as used by the existing MVA [75], is not enough to compute the queueing delay for a specific request. To improve the accuracy, our heuristic to the MVA considers the service demand for each individual access in the queue at a subsystem, and the numbers and types of requests in the queue at the time when this request enters. Such extension to improve the accuracy of the MVA has been independently proposed by Leutenegger [58] and others. Our contribution, in this thesis, is to model a multithreaded program workload such that the above mentioned extension to the MVA provides an accurate prediction of the system performance.

Given a multithreaded system and a program workload, we show how to derive the performance measures such as the processor utilization, the network latency for remote accesses with split-phase operations, and the message rate to the network. With architectural parameters and program workload characteristics, we characterize the variation in these performance measures.

1.4.2 Validation

We use simulations of the Stochastic Timed Petri Net (STPN) models as well as program executions on the EARTH system to validate our model predictions. Simulations of the Stochastic Petri Net (STPN) models, are commonly used for performance analysis [60, 81, 8, 23]. Motivations for our use of STPN models are as follows. First, under same assumptions as queueing network model, both techniques are equivalent and should yield same results. Second, STPN models can easily be extended to study complex interactions, which helps to assess the performance deviations of queueing network models under these conditions. Through simulations, we obtain the processor utilizations and latencies for network messages. Model predictions typically match within 10% of the simulation results.

To validate our performance model of the EARTH system [46], we execute synthetic

benchmark programs. We have developed a low overhead tool for a runtime measurement from actual program executions on the EARTH system. We measure the latency for split-phase remote accesses and the processor utilization. These measurements compare well within 5 to 20% of model predictions in most cases. Note that not all performance measures can be measured using software tools. Also, a small perturbation in the performance measures ($< 5\%$) occurs during runtime measurements.

1.4.3 Performance Analysis and Optimization

An analysis of the performance of all subsystems for each set of values for parameters provides us an insight to tune the performance of multithreaded architectures. First, such analysis points to critical values of parameters to achieve high performance. Second, the analysis shows the performance bottlenecks. Third, the analysis helps to assess how the changes to input parameter settings affect the processor performance.

Our performance characterization of multithreaded systems with model parameters shows critical values for which a high processor performance is achieved. *First,* for a processing node, we define the *effective memory latency* as the access time of the memory subsystem while servicing multiple, concurrent requests. To achieve a high processor utilization, the thread runlengths should be larger than the value of the effective memory latency. Thus, to support a fine grain program workload, a low effective memory latency is necessary. A use of interleaved memory banks and pipelined memory keeps the effective memory latency low. *Second,* for a remote memory access pattern, we define the *network capacity* as the maximum message rate per processor delivered by the network. The processor utilization improves with an increase in the number of threads, as long as the message rate is not close to the network capacity. The processor performance increases despite the increasing network latencies and message rate. *Third,* we investigate the effect of thread characteristics like the number of threads, their runlengths and the number of outstanding requests during each runlength of a thread. We show that the network latency increases more rapidly with the number of outstanding requests per thread than with the number of threads. Also, the higher the number of outstanding requests per thread, the lower the processor utilization. This decrease in the processor utilization cannot be compensated by increasing the number of threads. Instead, the performance improves with optimizations

like increasing the thread runlength.

We introduce a metric, the *tolerance index*, to quantify the effectiveness of multithreading to tolerate latencies at a subsystem. The latency for an access is *tolerated*, when the processor does not idle due to this access. The tolerance index, say for the network latency, shows how close the system performance is to that of an *ideal* system, which incurs no network delays. For performance tuning, the lower the tolerance index, the greater the possibility of gains due to performance optimizations. Thus, a user can analyze the tolerance of latency at individual subsystems, and tune the program workload with respect to these subsystems.

We apply our model to analyze the performance of McGill's EARTH multithreaded multiprocessor system. The model predicts how various thread characteristics affect the performance of a multithreaded system (like the EARTH) using realistic costs of multithreading operations. Measurements from program executions on the EARTH system match well with the analytical model (within 5 to 20% of model predictions). While we use synthetic benchmark programs to study impact of individual thread characteristics, we also show applications of the model to optimize real benchmark programs. Our results from the EARTH system demonstrate the tradeoffs of realistic costs of multithreading on the performance for fine-grain parallel program workload. For example, on current implementation of the EARTH system [46], programs yield processor utilizations above 80% when 4 to 8 threads have more than runlengths 3000 cycles and a low (≤ 3) number of outstanding requests per thread.

We also explore how the changes in implementation of the EARTH system will affect its performance. Specifically, we study how much performance benefits are obtained using multithreading if costs of an EARTH processing node are reduced by 50%. Similarly, how much performance gains are possible if the EARTH system employs a slower network (like in a *NOW*, network of workstations).

Through examples, we show how to apply our analytical results to optimize the program workload characteristics to achieve high performance. For users of multithreaded architectures, our analytical results provide an insight to the impact of performance related optimizations in the presence of long latencies on real systems.

1.5 Scope of the Performance Analysis and Tools

This section provides the details on the generality of our approach and the scope of the tools developed in this thesis.

1.5.1 Modeling and Analysis

Our analytical model is developed using dataflow-based multithreaded program executions, i.e. the threads are atomic. However, our model is applicable to the von Neumann style multithreading as illustrated by an example in Chapter 2 and experiments in Chapter 8.

The multithreaded program execution model is well-suited for a Single-Program-Multiple-Data (SPMD) model of computation [43]. The SPMD model has been widely successful on distributed shared memory machines, and provides users with a tangible set of parameters to characterize the parallel program workloads. Since a performance characterization of other parallel program structures like recursion (e.g. Fibonacci) is not general, this thesis does not consider these program structures. However, we believe that for multithreaded workloads, the optimization hints obtained from an SPMD model are good heuristics to tune other parallel program structures as well.

We have developed a solution package to analytically solve the closed queueing network models of multithreaded systems– uniprocessors and multiprocessors. Initially, we focus on simple architectures of processing nodes to explore the benefits of multithreading. We model simple interactions among the subsystems, like split-phase memory accesses. Then, we apply the analytical model is to analyze the EARTH-MANNA system. This application shows how to account for realistic subsystem interactions at an EARTH-MANNA node, and how much is their impact on the performance of the system. We focussed on single-data accesses in the EARTH-MANNA system, however, a similar extension of the model is possible for other subsystem interactions, like block-data transfers in the EARTH.

We have analyzed two distributions for data locality, *geometric* and *uniform*. Chapter 5 discusses how to extend our approach to analyze other data distributions.

1.5.2 Performance Tools

Apart from the solution package for our analytical models, we have also developed a simulator and measurement tools to validate the model predictions.

To validate the model predictions, we measure the performance of the EARTH-MANNA system. A simple instrumentation of the application program allows us to measure the network latency with an accuracy of 5 cycles. This instrumentation gathers latency values on any application, where a processing node can be dedicated for a *measurement* thread. The dedicated processing node executes measurement thread to monitor remote accesses (Chapter 7 provides the details). The latency is measured for one access at a time. The processor utilization is measured using a function which counts the cycles when there is no computation thread to execute.

A simulator is developed to validate model predictions for abstract uniprocessor and multiprocessor multithreaded systems. The simulator is written in Voltaire [72], a language to specify the net-list for colored petri nets. A significant processing at each place and transition, based on attributes of the tokens, provides the following flexibility. Detailed configurations of architectures can be quickly simulated, and a wide range of performance statistics can be easily gathered. We have simulated a crossbar and a mesh network which is modeled as a petri net with 80 places for a 16 processor system. Each processing node contains 10 places in the petri net, as shown later in Figure 5.5. The simulator for a 16 node machine takes 1 to 5 minutes for each run on a SPARC-10 workstation. (The analytical prediction requires less than 1 minute for a 16-processor system.) The simulator can execute, a steady state pattern (used in this thesis), as well as a parallelism profile based pattern using synchronizations. We use deterministic and exponential service time distributions at various transitions. However, the simulator allows other statistical distributions.

1.6 Synopsis

This thesis is organized as follows. In the next chapter, we survey the existing multithreaded architectures. Using a program workload, we show how to achieve an overlap of computation and communication. We classify the existing work on performance evaluation of multithreaded architectures as: performance modeling, simulations, and system

measurements.

In Chapter 3, we discuss the issues in performance modeling, analysis and optimizations of multithreaded architectures. We define the statements of the problems solved in this thesis. We outline our approach to evaluate the performance of multithreaded architectures. We also describe the multithreaded program execution model for which analytical models of single and multiprocessor systems are developed in later chapters.

In Chapter 4, we propose an analytical model to predict the performance of a single processor multithreaded system. Our analysis shows how the organization of a multithreaded processing node affects its performance. We also discuss implications of these results for performance optimizations.

In Chapter 5, we propose an analytical model of a multithreaded multiprocessor system. We validate model predictions using simulations of a stochastic timed petri net model of the system. We show how to derive key performance measures of interest, and characterize the variation in these performance measures using critical architectural and program workload parameters. We show the impact of performance optimizations on the performance of system resources like memory and network switch.

In Chapter 6, we discuss what we mean by the latency tolerance, how to quantify the latency tolerance, and how does it help in performance optimizations.

In Chapter 7, we extend our analytical model to analyze the performance of the EARTH multithreaded multiprocessor system. We propose a simple solution under multithreading to the problem of *simultaneous resource possession*. Measurements from the actual program executions on the EARTH system validate the model predictions.

In Chapter 8, we also characterize the performance of the EARTH system under realistic costs for multithreading. We discuss how program optimizations as well as changes in system configurations affect the the performance of the EARTH system.

In Chapter 9, we present an overall perspective of this thesis, and outline future directions of this research.

Chapter 2

Background

The objective of this chapter is to familiarize the reader with how multithreaded architectures operate, what problems do they pose for their performance measurements, and what are the existing methods for their performance evaluation. This background will provide an insight to the issues in performance modeling and analysis of multithreaded program executions, as discussed in later chapters. In Section 2.1, we describe the underlying architectural mechanisms in existing multithreaded architectures. Through an example of a program workload in Section 2.2, we show how a programmer can use multithreaded (*split-phase*) operations to achieve an effective overlap of computation and communication. In this thesis, this workload is used as a running example to show performance optimizations on multithreaded architectures. In Section 2.3, we discuss the problems in the measurement of performance of multithreaded systems. In Section 2.4, we survey recent studies on performance evaluation of multithreaded architectures. We categorize these studies as the analytic performance modeling, simulations, and system measurements.

2.1 Mechanisms in Multithreaded Architectures

In Chapter 1 (Section 1.1), we outlined a multithreaded program execution model. Now we will discuss how a multithreaded architecture supports such a program execution model.

Two essential features of multithreaded architectures are: a mechanism to issue split-phase transactions like a remote memory access, and another mechanism to rapidly switch

to execution on any of the available threads. During the execution of a thread, when a processor encounters a long latency split-phase transaction, the processor issues the access and rapidly switches the context to execute on another thread. So, the overall idle time at the processor reduces. For example, let us consider the progress of computation on two threads at a processor in Figure 2.1. Each thread occupies the processor only for R time units, and idles for L time units till its memory access is serviced. Note that despite incurring an overhead of C time units for each context switch, the overall idle time (shown as empty boxes) at the processor is reduced.

Above mechanisms to support a multithreaded program execution model can be implemented in the hardware, e.g. the processor of TERA [9] and April [6], or in the software, e.g. a run-time support in TAM [26], *T [68] and EARTH-MANNA [46].

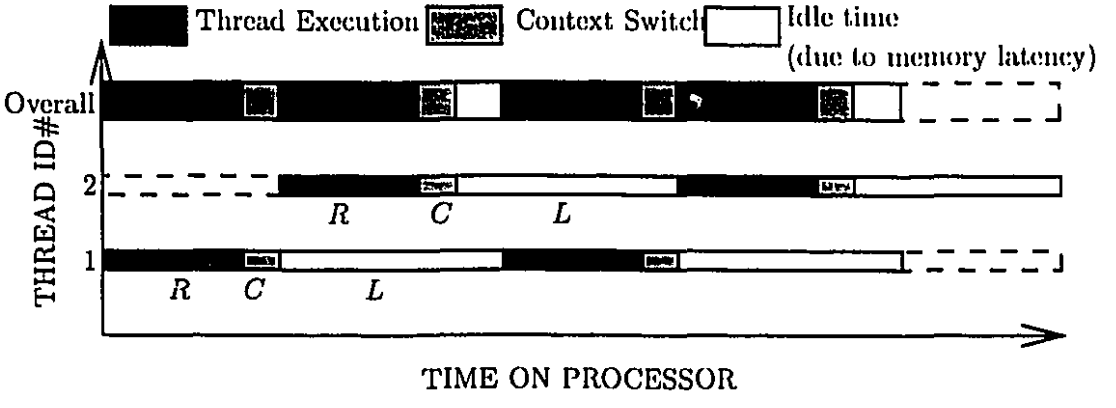


Figure 2.1: Computation and Communication Overlap in a Multithreaded Processor

Hardware Mechanisms in a Processor:

A multithreaded operation is a split-phase operation with phases like sending remote messages, accessing local memory for a read/write request. Hardware mechanisms use the processor and surrounding circuitry, to detect these split-phase operations and service individual phases. On completion of a multithreading operation, the execution on the corresponding thread may proceed. Typically multithreaded processors maintain contexts for multiple threads in its register set(s). So, a context switching consumes a small time, e.g. one cycle on TERA [9] and 14 cycle on April [6]. Off-the-shelf microprocessors, however, need modifications to support these hardware mechanisms.

Three mechanisms to support this execution are:

- *Cycle-by-cycle interleaving:* A processor concurrently executes on multiple threads. In each cycle, the processor switches context to a thread selected in a round-robin manner. If the memory access issued by a thread has not returned, the processor idles at the turn (cycle) for that thread. TERA [9] and its predecessors– HEP [87] and Horizon [93]– adopt this approach.
- *Block multithreading:* A processor executes on one thread till it encounters a cache miss on a *remote memory access*. The remote access is issued, and the processor switches context to another thread. The context switch time is typically over 10 cycles. Once the remote access is complete, the thread is ready for execution. Processors in Alewife [5] and [100] adopt this approach.
- *Hybrid scheme:* This scheme combines the advantages of above two approaches. A processor executes on a thread till it encounters an *off-chip access* i.e. a local or remote cache miss. The processor switches to another thread in one or two cycles. When an access is completed, the execution on corresponding thread can continue. This approach is adopted by processors in [51, 40, 55].

Software Mechanisms in a Run-Time System:

These mechanisms focus on the use of traditional multiprocessor systems. Multithreading primitives to invoke these mechanisms are supported in high-level languages, e.g., EARTH Threaded-C [46] and Spilt-C [25]. At compile-time, these primitives are treated as a function call or expanded/in-lined as an assembly language subroutine. Since the run-time system handles these mechanisms, multithreading primitives provide the entry and exit points of the run-time system.

When a thread issues a split-phase transaction to fetch (or store) a remote data, the run-time system allows the processor to continue the execution on other threads. The run-time system ensures that the completed remote access is returned to the waiting thread, and the ready thread is scheduled for execution. This approach is used in EARTH [46], TAM [26], and *T [68]. As an example, let us consider a remote memory fetch operation, `GET_SYNC` operation, on McGill's EARTH system.[1]

[1] We refer to McGill's EARTH-MANNA system as the EARTH system in this thesis. McGill's EARTH architecture [46] is currently implemented on the MANNA system developed by GMD, Berlin, Germany [20].

(Appendix E outlines these primitives.) When a processor encounters a `GET_SYNC` operation, the processor executes the corresponding assembly language subroutine. This subroutine places a message for other functional unit in the processing node (synchronization unit, SU), and returns the control of computation to the point after the `GET_SYNC` operation. Thus, a split-phase operation is issued. When a remote access is complete, the SU at the local processing node activates the thread waiting for this data. A thread may require one or more remote accesses to complete before the start of its computation. On completion of these remote accesses, the SU schedules this thread for the processor to execute on. When the processor changes context at a later instant, this thread is ready for further progress in computation. (The details of the operation of the EARTH are discussed in Chapter 7.)

In comparison to hardware mechanisms described above, a context switch on the EARTH system takes at least 36 cycles. Even though only 6 to 7 instructions are required for a context switch, the main cost is due to two cache misses on average to save at least two registers in the local memory. The advantage of software mechanisms, however, is that off-the-shelf microprocessors can be used without any expensive modifications to their designs.

The mechanisms discussed above are specific to multithreaded systems. In addition, a support is needed for following thread operations: *thread creation*, *data communication* and *thread synchronization*.

Thread creation: A thread is created to perform a computation task. Each thread has a unique identifier.

Thread synchronization: A thread can synchronize with one or more threads. When these threads synchronize, a predefined state of computation is reached. A message is sent for a synchronization when either the threads communicate with each other or certain threads complete their computation tasks.

Thread communication: A thread communicates with another thread through data values for shared variables. A synchronization message ensures that correct values of shared variables are communicated.

These three mechanisms are also found on single threaded multiprocessor systems to support a concurrent execution on multiple processors. Two major differences, however, exist between multithreaded architectures and single threaded multiprocessor architectures. First, multiple threads can be concurrently active on a multithreaded processor, so thread identifiers are needed to perform most of the above operations. In contrast, an identifier for a single threaded processor serves the purpose of identifying the thread it executes. Second, a *thread scheduling* mechanism is needed to select the next thread for execution, save the context for the current thread, and restore the context for the next thread. The earlier discussion in this section shows that a mechanism to switch the context is provided either in the hardware (of a multithreaded processor) or in the software (through a run-time system).

So far we discussed some basic mechanisms to support multithreading. Now, we will focus on their use in a program execution.

2.2 A Multithreaded Program Workload

In this section, we show a simple example on how to effectively use the multithreading operations. The objective is to achieve an overlap of computation and communication such that the processor performance improves. We use multithreading primitives in EARTH Threaded-C [46] for our discussion, and elaborate them as we encounter.

Let us consider an addition of two 2-dimensional matrices, i.e. $C = A + B$. The pseudo-code is shown in Figure 2.2. In the `MAIN` body of the program, lines 24 through 26 indicate how threads are forked on P nodes in the system. At the end of computations, all threads synchronize and a thread with label `TREAD_complete` is triggered (see line 27). Using the function `INVOKE` on line 25, we create P copies of `new_thread` on P processing nodes of the multithreaded multiprocessor system. After forking these threads from node 0, `END_THREAD` is used to switch the context to any thread ready for execution at node 0.

On every processing node, `new_thread` spawns n_t `compute` threads (lines 15 to 17). After execution, these n_t `compute` threads return the control of computation to `THREAD_done` (line 18). A `compute` thread performs additions for K elements (line 3). For each addition, two remote memory fetches, `GET_SYNC` operations, are issued in parallel (lines 4 and 5). Since

these are split-phase long latency transactions, the processor can be freed for an execution on other threads. So, we use an END_THREAD operation for the context switch (line 6). On the return of GET_SYNC operations, the execution on THREAD_add can begin (line 7). The result of the addition is stored using one DATA_SYNC operation (line 9). Note that all the three remote data accesses (two GET_SYNCs and one DATA_SYNC) are independently executed. Their completion, however, is necessary to trigger THREAD_add. A small additional code is necessary at the start and end of computation shown between lines 3 and 9. At the end of computations, thread synchronizations follow an hierarchy similar to their forking.

Figure 2.4 illustrates an abstraction of the progress of computation at various nodes for the pseudo code shown in Figure 2.2. There are five types of threads shown in Figure 2.4:

A: Thread A forks threads on different processors. Lines 24 to 26 in Figure 2.2 represent a thread A.

B: Thread B forks multiple threads on the local node for computation. The new_thread in Figure 2.2 is of the type thread B.

C: Thread C performs computations on the local processor, sends and receives long latency accesses, and finally sends a completion signal (to a thread D) to synchronize. The compute thread in Figure 2.2 is of the type thread C.

D: Thread D collects synchronization signals from threads in the local processing node. In Figure 2.2, D thread is not explicitly shown. However, on completion of the thread D, the execution reaches line 18.

E: Thread E collects synchronization signals from threads on different processing nodes. In Figure 2.2, E thread is not explicitly shown. However, on completion of the thread E, the execution reaches line 27. Results of the loop structure described by Figure 2.2 are correctly visible at line 27.

Thus, a thread A forks one copy of thread B on each processing node. Each thread B forks multiple copies of thread C for computation. At the completion of computation, C threads synchronize locally and initiate the thread D. Finally, D threads from all nodes synchronize at the thread E.

```
1: THREADED compute (parameters...)
2: {                                    /* j-th thread at node i */
3:     for (k=low; k < high; k++)       /* say low=0 and high = K */
4:   { GET_SYNC (A[k][0],a[k],add);     /*Synchronize at THREAD_add.*/
5:     GET_SYNC (B[k][0],b[k],add);
6:     END_THREAD ();                   /*Context switch. On fetching, go to THREAD_add.*/
7:   THREAD_add:
8:     c[k] = a[k] + b[k];
9:     DATA_SYNC (c[k],C[k][0],add);
10:  }
11:                                     /* Thread completed.  Return to new_thread */
12:}
13:THREADED new_thread (parameters...)
14:{                                    /* at node i */
15:  for (j=0; j < n_t; j++)
16:  { SPAWN (compute,j,done); }        /* Fork n_i compute threads at node i */
17:  END_THREAD ();                     /* Switch.  Barrier.  Go to THREAD_done.*/
18:  THREAD_done:                      /* All C values are computed */
19:                                    /* Return to node 0 */
20:}
21:THREADED MAIN()
22:{ ...
23:  /* at node 0 */
24:  for (i=0; i < P; i++)
25:  { INVOKE (i,new_thread,complete); }  /* Fork i-th thread on node i */
26:  END_THREAD ();                     /* Switch.  Barrier.  Go to THREAD_complete.*/
27:  THREAD_complete:
28:                                    /* All processors have synchronized */
29:  ...
30:}
```

Figure 2.2: A Multithreaded Program Workload

The maximum time in a program execution in Figures 2.2 and 2.4 is typically spent in thread C, i.e. compute threads. A support for multithreading techniques described in Section 2.1 ensures that threads A, B, D, and E are executed efficiently (we will show measurements from the EARTH system in Chapter 8). At steady state, n_t compute threads are in various states of computation at each processor: *ready*, *executing* or *suspended*. A thread is *ready* for execution by the processor when all input operands are available. Once scheduled on the processor, the thread is in *executing* state. A thread is *suspended* when it is waiting for its operands. Figure 2.3 shows the states each compute thread experiences between the iterations k=low and k=high-1 (lines 3 to 9). *This steady state behavior of a multithreaded program execution is the primary focus of our performance modeling and analysis.*

The execution time on the EARTH system for this program is shown in Tables 2.1 and 2.2. The matrix size is for 32768×8 elements. Cache organizations affect the number of read/write accesses. To ensure a constant number of read/write accesses in each program execution, we use 8 elements in each row of matrices and access the first element of each row. Table 2.1 shows the execution time when all GET_SYNC and DATA_SYNC operations are sent uniformly to all nodes in the system. With a use of multiple threads, the idle time on each processor and the program execution time reduces. The speedup at larger number of processing nodes is poorer. For this program, we increase the computation between lines 7 and 9 such that the thread runlength is 3000 cycles. Table 2.2 shows the execution time for this example. Now, a linear speedup is achieved with respect to the number of nodes in the system. With multiple threads, the performance improves by as much as 25% on a 16-node system.

In this example, we increased the runlength of each thread by 60 times, i.e. increased the computation per thread. With one thread on one node, the increase in program execution time was only 7.08 times ($=\frac{2273}{321}$), and on 19 nodes by only 49% ($=\frac{145-92}{92}$). Also with 19 nodes and each having 8 threads, the execution time at higher runlength increased by merely 35% ($=\frac{120-89}{89}$). An objective of the performance evaluation is to find out whether these trends continue with number of threads and number of processing nodes, is there an optimal thread granularity given a number of processing nodes, and which program workload characteristics can be optimized. Even for simple loops discussed above, such analysis is necessary, because contentions due to multiple accesses from each processing

node increase significantly in a multithreaded system.

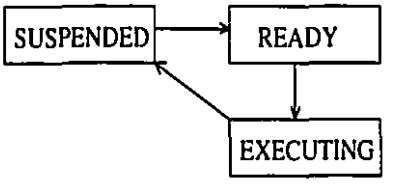


Figure 2.3: States of a Thread

In this thesis, we model and analyze the steady state behavior of a multithreaded execution of the program workload in Figure 2.2. This workload is used as a running example for performance optimization purposes.

2.3 Issues in Performance Measurement

The execution time of a program workload is the easiest measurement of the performance of a computer system. The execution time, however, yields little information on how to optimize a program workload. Performance measures of our interest are, the processor utilization, the message rate to the network, and the network latency (i.e. the latency for multithreading operations like GET_SYNC on EARTH system [46]). These performance measures pose difficulty in the runtime measurements on a multithreaded system. The objective of this section is to discuss the problems in their measurement on multithreaded systems, and outline our approach for the EARTH system.

Note that detailed simulations of multithreaded systems also provide the above performance measures. However, simulations, like other performance prediction techniques, are a representative of the realistic performance behavior only for the ranges they have been validated.

Tools for obtaining the performance measures from program executions on a system are invaluable to the task of performance tuning. Important aspects in the choice of tools are, the ease of use, the accuracy of measurement, and the perturbation to the system execution. As we mention below, the message rate is easy to measure, however the network latency and processor utilization require specific considerations on multithreaded systems.

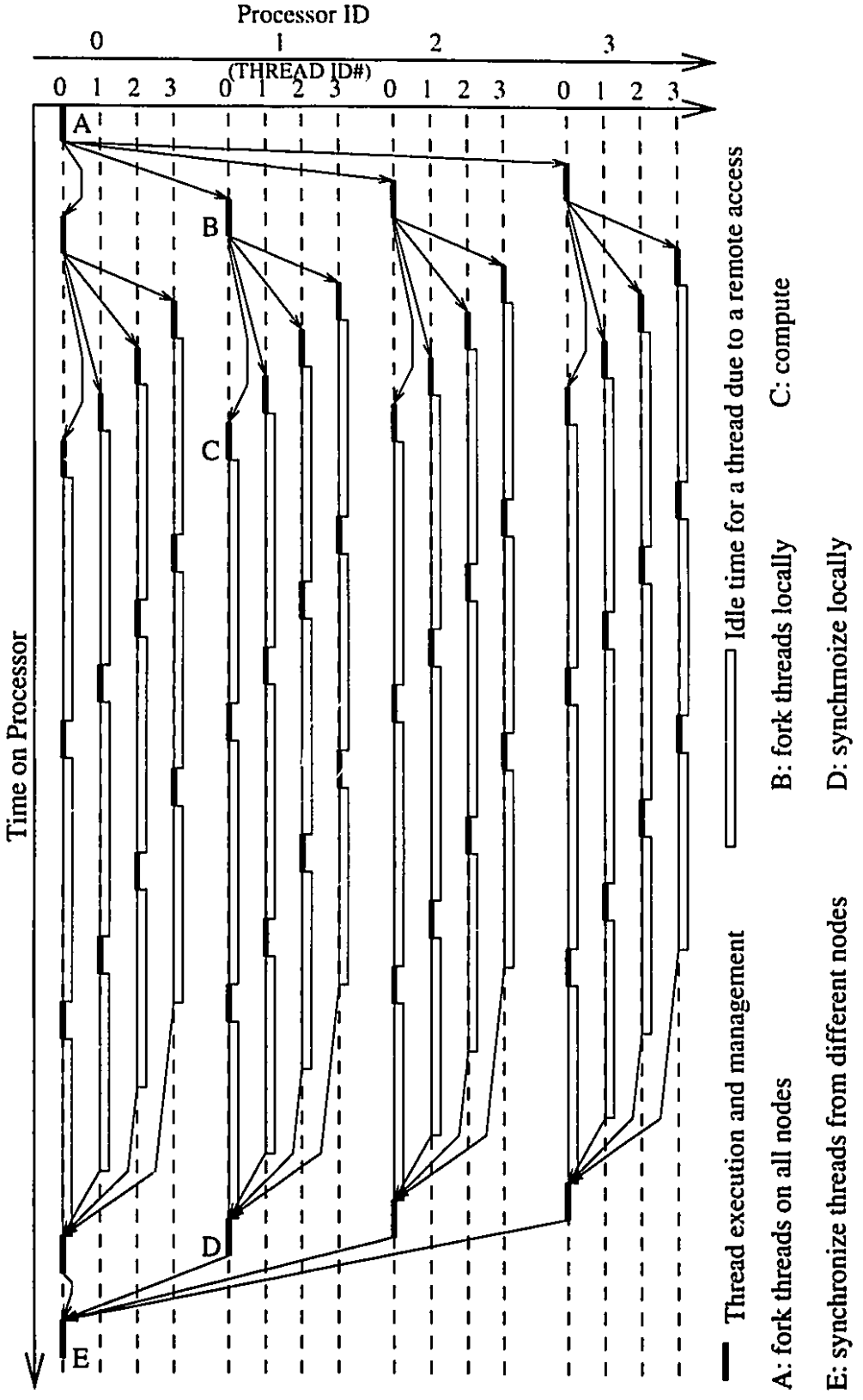


Figure 2.4: An Abstraction of a Multithreaded Program Execution

Number of Threads	Number of Nodes					
	1	3	7	11	15	19
1	321	227	129	109	97	92
2	254	166	115	102	94	90
4	241	155	113	100	93	89
8	241	161	112	100	93	89

Table 2.1: Execution time in *ms*. Runlength = 50 cycles. Uniformly distributed data accesses.

Number of Threads	Number of Nodes					
	1	3	7	11	15	19
1	2273	846	372	244	180	145
2	2175	747	324	208	154	122
4	2168	724	310	200	146	116
8	2170	720	309	199	145	120

Table 2.2: Execution time in *ms*. Runlength = 3000 cycles. Uniformly distributed data accesses.

The measurement of network latency poses difficulty on a multithreaded system for the following reason. The multithreading technique overlaps the communication on one thread with the computation on another. So, the processor may switch to execute on other threads after initiating the split-phase communication access on one thread. When the response arrives, the processor may be busy executing on other threads. Thus, the processor cannot measure the precise elapsed time between the initiation of a remote access, and the arrival of its response. There are following two choices for measuring the latency:

1. *Direct measurement (Time-stamp)*: This method makes use of a time-stamp on a message to measure the network latency. Overheads involved in this method are: the maintenance of a timer, the increase in the message length to include time-stamp, and the maintenance of the network latency statistics. These overheads perturb the measured value as well as the program execution for the following reasons. To compute the latency and store it in the memory, certain number of instructions and memory

accesses are required. Further, accessing the timer, updating its value, and placing a time-stamp on a message, requires additional instructions and memory accesses. Together, these additions can significantly alter the runlength of a thread as well as the execution time on the processor. Similarly a change in the message length changes the service time for a message on network switches and alters the latency. For fine grain applications, to exploit an overlap of computation and communication, thread runlengths are typically within an order of magnitude of the network latencies (we will discuss more in Chapter 4). So, overheads in the *direct measurement* can easily change the program workload characteristics and the performance measures.

2. *Indirect measurement (Sampling)*: In this method, a thread is dedicated to the task of measurement. This thread requires an exclusive access to a processor during the measurement. We refer to this processing node as the *dedicated* node, and the nodes under investigation as the *test* nodes. The thread sends test messages to test nodes, and measures the time till the receipt of their responses. The latency measured by this method indicates how much contention a message suffers during one round-trip. An exclusive access to the dedicated node for time measurement ensures that the dedicated node does not delay the test message.Any delay, in excess of the no-load value of the latency, is a result of contentions at the network and the remote node(s). A drawback of this method is as follows. When more than one remote messages are sent by a thread, the arrival of the first response will stop the measurement, since the processor gets busy with the latency computation. Thus, an accurate timing for the subsequent arrivals cannot be known. An advantage of this method is a runtime measurement of latency with very little overhead.

In this thesis, we developed a software instrumentation with the *sampling* approach to measure the network latency on the EARTH-MANNA system. The accuracy of the tool is 5 cycles (i.e. 100ns on 50 MHz Intel i860 XP). The tool provides a flexibility to be turned on or off dynamically during a program execution. The tool introduces less than 2 to 5% increase in the program execution time even for communication intensive programs.

Our approach to measure the processor utilization uses the measurement of idle time at the processor. Whenever there are no threads in *ready* queue to execute on, the processor executes a function to measure the idle time. As soon as a thread is ready for execution,

the idle time measurement terminates. Two inaccuracies involved in this measurement are as follows: *First*, switching to and from the function to measure the idle time incurs an overhead. Currently, the measurement function does not record the number of times it is invoked, so a correction for this value cannot be performed. *Second*, the idle time for a processor, while waiting for the local bus accesses, cannot be measured in software. Thus, a higher processor utilization is reported.

The message rate to the network is computed by counting the number of remote accesses, measuring the execution time for a program and obtaining their ratio.

On the EARTH system, the program execution time can be measured with an accuracy of 2.5μs. Above measurements for processor utilization and message rate introduce up to 5% increase to the program execution times as small as 100 ms.

2.4 Related Work

Previous sections discussed the mechanisms for multithreaded execution, their use in a program workload, and the problems they pose in the performance measurement. In this section, we briefly outline how various studies have carried out the performance evaluation. Most of these studies are on multithreaded architectures, and can be classified as: analytical models, simulations, and system measurements. We discuss detailed differences these work with our contributions in this thesis at the end of relevant chapters.

Analytical Models:

These models are further classified into following two categories.

Queueing Network and Petri Net Models:

Saavedra et al [80] proposed the first analytical model based on Petri Nets to predict the performance of a multithreaded processor. This simple model is based on only four parameters– the runlength and number of threads representing the workload, and the context switch time and memory latency representing the architecture. They use state-space analyses to derive the system performance, so modeling a multiprocessor system and the contentions at subsystems, is computationally expensive. (That is, not possible for as small as 16-processor systems.)

Alkalaj and Bopanna [8] proposed a Petri Net based model for a dynamic multithreaded workload on a bus-based multiprocessor system. However, the bus contentions are not considered. Also, the applicability of a dynamic program workload model is not clear. The solution is based on the state space of the petri net model, so an extension to incorporate contentions is computationally expensive.

Yamamoto et al [103] proposed an analytical model for a superscalar multithreaded processor. Their model accounts for the data and structural hazards during the execution on an instruction. They focus on achieving high instruction level parallelism, in the absence of contentions at the memory.

Torrellas et al [94] proposed an open queueing network based model for a DASH-like single-threaded multiprocessor system. They characterize the performance variations for small variations in input parameters. An extension to multithreaded system does not appear straight-forward.

Willick and Eager [101] proposed a closed queueing network model of an interconnection network which supports multiple outstanding requests per processor. The system behavior is similar to a multithreaded system. Adve and Vernon [2] proposed a closed queueing network model for multithreaded multiprocessor systems with k-ary, n-cube interconnection network. Solutions to these models use mean value analysis, a computationally efficient method, especially for analyzing large systems. These closed system models capture the system behavior more realistically, as discussed later in Chapter 5.

Analytical models in [101, 8, 66, 2] are validated through the simulations of petri net and queueing network models. Note that due to lack of details on simulated systems in these work, we assume that discrete event simulations are used, and that their assumptions are similar to those for respective analytical models. Saavedra et al [80] report a validation using simulation results (from [100]).

Cache Parameters Based Models:

Agarwal [4] proposed an analytical model based on cache parameters, for a multithreaded processor. The proposed model characterizes additional cache misses due to a multithreaded execution on the processor. The model does not include the feedback effect of network performance on the cache miss rates. Johnson [50] extended

this model to incorporate the network performance. We will discuss advantages and disadvantages of using Johnson's approach in Chapter 5. Both models are validated using simulations of the Alewife system [5].

Simulations:

Simulation of a multithreaded program execution is an aid to verify the correctness of the program behavior as well as the architectural design, and to evaluate the performance realistically. A typical simulation approach imitates an execution of a program workload on a behavioral model of functional units in the multithreaded system. Performance measures are computed by maintaining the relevant statistics (like idle time of the processor, time for an access to complete etc.). The accuracy of a performance prediction depends on how detailed is the simulation model, e.g. whether contentions at subsystems are modeled. Advantages of simulations are that the effect of realistic program executions on the system performance is captured. Changes to the system design as well as the program workload can be studied. However, with detailed modeling, only small program sizes can be simulated. Two approaches for simulations are typically used: *trace-driven simulations* and *system simulations.*

Trace-driven Simulations:

These simulations use address traces and model the subsystem interactions. By not maintaining the state of computations, these simulations reduce the complexity, and improve the speed. However, the address traces are normally generated from single-threaded multiprocessor executions [100, 90]. The execution trace of one processor is treated as a single thread. Multiple copies of a single processor trace form multiple threads of computation on a processor. Artificial synchronization points are inserted to study the effect of various program characteristics like sharing of data variables and synchronization on the performance.

Weber and Gupta [100] performed trace-driven simulations for a shared bus system with strategies for switching contexts, and constant context switching times. They considered a workload with multiple copies of a program trace as multiple threads. Thekkath and Eggers [90] extended a similar approach using an analytical model [3] for the network performance. Waldspurger and Weihl [98] report the results of simu-

lations on a single node of multiprocessor system. They also assume that the network is lightly loaded, i.e. no contentions.

System Simulations:

System simulations focus on the correctness of program executions as well as the subsystem interactions [49]. So, a large state of the system is maintained at each cycle (or event). However, the speed of simulation decreases, and large benchmark programs take very long time to execute. To improve the speed of simulations, behavioral models of subsystems are used. These behavioral models may not accurately capture the operations of functional units, and are potential sources of errors in performance prediction.

Alewife [6, 5, 50] has been extensively studied using simulations. For McGill's EARTH system, a simulator SEMi-a Simulator for EARTH, MANNA and i860- is being developed [92]. The objective is to exhaustively study the program executions and potential bottlenecks of the EARTH system. To speedup the simulation of a large EARTH system, the simulator runs on the EARTH system itself.

System Measurements:

Actual measurements, from program executions on a system, provide an accurate measure of the performance. Hardware probes as well as software subroutines are useful for performance measurements. However, during multithreaded program executions, difficulties arise in measuring the latencies for accesses as well as various subsystem delays (as outlined in Section 2.3). Alewife [5] and EARTH [46, 64] are two multithreaded systems for which performance measurements have been reported.

Arpaci *et al* [13] report a characterization of latencies for various operations on CM-5 using Split-C. A similar evaluation of a multiprocessor system using synthetic benchmarks is reported by Boyd and Davidson [19]. These studies are indeed very useful for compilers and programmers to choose which read/write features to use in an application. However, they do not consider the impact of overlap of computation and communication on performance measures.

In this thesis, we propose analytical models for single processor and multiprocessor multithreaded systems. To model and analyze realistic architectural interactions and pro-

gram workload on McGill's EARTH system, we expand the set of parameters. We validate the analytical mode predictions using simulations of petri net models and measurements from the EARTH multithreaded multiprocessor system. We analyze the effect of changes in program workload characteristics as well as architectural parameters on the system performance. These are significant extensions over previous studies. By addressing above issues, we believe that our work provides a strong evidence on the usefulness of the analytical models for performance optimizations on multithreaded systems.

2.5 Summary

In this chapter, we discussed the hardware and software mechanisms to support multithreading technique in various computer architectures. The handling of split-phase operations, and the management of contexts for threads are the key to an efficient multithreading support. We illustrated how a user (programmer) can use the multithreading operations, *fetching a remote data, context switching,* and *storing to a remote location,* to achieve an effective computation and communication overlap. This improves the processor performance. We showed how this overlap poses problems in performance measurements on multithreaded systems. Then, we surveyed the literature for performance evaluation of multithreaded architectures. Most of the studies revolved around trace-driven simulations, and analytical performance modeling. This thesis departs from the existing literature in the following way. First, we propose analytical performance models of multithreaded systems– single processor, and multiprocessor systems. We show how to incorporate realistic subsystem interactions through a case study on McGill's EARTH multithreaded multiprocessor system. Second, we validate the model predictions using performance measurements from actual program executions on the EARTH system. Third, we show how the performance models can be used for performance related optimizations of the architecture as well as the program workload.

With above discussion on the hardware and software issues of multithreaded architectures, and their performance evaluation, we are ready to explore the performance modeling, characterization, and analysis of multithreaded architectures.

Chapter 3

Problems Statements for Performance Analysis

Previous chapter outlined basic mechanisms to support multithreading techniques and their use in a program workload. We also discussed issues in performance measurement on multithreaded systems, and surveyed existing approaches.

The objective of this chapter is to state problems on the performance modeling and analysis of multithreaded architectures studied in this thesis. These problems focus on our approach to the performance prediction, performance analysis, and usefulness to users of multithreaded systems.

In Section 3.1, we discuss the challenges to the performance modeling of multithreaded architectures. In Section 3.2, we define the statements of problems on the performance prediction solved in this thesis. In Section 3.3, we outline our approach to address these problems. In Section 3.4, we describe the multithreaded program execution model. In later chapters, this program execution model serves as a basis to develop analytical performance models of single processor and multiprocessor multithreaded architectures. In Section 3.5, we summarize the discussion in this chapter.

3.1 Performance Modeling Issues for Multithreaded Architectures

This section discusses how the performance modeling of multithreading differs from that of traditional systems, namely, single-threaded architectures and multitasking operating systems.

A multithreaded processor supports multiple outstanding accesses. In contrast to a single-threaded system, these accesses can simultaneously keep multiple subsystems busy. The performance modeling of multithreaded architectures differs from that of single-threaded architectures as follows:

1. A processor can continue the execution (on another thread) after issuing a long-latency memory request (to its local or remote memory). After the memory access request is serviced for a thread, the processor may not execute on the thread immediately.

2. Accesses from multiple threads contend at subsystems, increasing their observed latencies (even at local memories). In turn, longer latencies for individual accesses delay the execution on waiting threads.

While multithreading helps to increase the processor utilization, the increased contention reduces it. To weigh this trade-off, a complete performance model should capture the feedback effect of the concurrent activities at various subsystem resources on the rate of accesses to a subsystem and latency of a subsystem.

Similar to the multithreading, operating systems use multitasking technique to improve the throughput of a system by assigning time slices to multiple tasks. When a task accesses secondary memory or waits for interactive response from the user, an idle time results at the processor subsystem and primary memories. The multitasking technique schedules the time-slice for the execution of a task on which useful work can begin immediately. In general, multiple tasks do not co-operate on the same application program. Unlike multithreading, the overheads of multitasking are significantly larger than typical communication latencies observed on the interconnection network. The presence of multiple tasks does not affect the communication latencies for individual remote memory accesses on the network. Thus,

the following two characteristics of multitasking make its performance modeling different than that of multithreading:

- During a time-slice, one task has an exclusive access to the processors and primary memories. Accesses from different tasks do not contend at primary memories and network. So, the latencies are similar to single-threaded architectures (i.e. close to their no-load values).
- Queueing delays and contentions due to multiple tasks are encountered only for accesses to shared resources like disks and secondary memories, and to gain access to a processing subsystem with primary memories. The execution time of a task on the processing subsystem is assumed to be independent of the low level computation and communication. That is, a typical modeling of multitasking technique does not involve a detailed program execution with individual accesses to local and remote memories in a multiprocessor system.

A multithreaded program execution on real systems causes additional complications due to interactions among various subsystems. In our case study of the EARTH system, we have explored the following two problems on performance modeling. These are representative of problems one often encounters on other real systems as well.

- *Simultaneous resource possession*: On a system like the EARTH, resources at a processing node can be accessed only through the bus. Further, the bus is held till the access completes. The challenge is to predict the waiting time at each resource accurately. In queueing theory literature, this problem is known as *simultaneous resource possession* [48, 56]. Access contentions at such resources increase significantly in the presence of multiple outstanding requests per processor.
- *Multithreaded program workload*: Multithreaded program workload may exhibit different thread characteristics at different processing nodes. The overheads of multithreading operations can be high, so it is essential to accurately characterize the program workload, and to accurately compute the queueing delays.

In summary, multiple outstanding requests increase the severity of the contention problem at system resources. The consequences to a user of multithreaded architectures are as

follows. The no-load values are no longer a good indicator of the subsystem performance. So, program workload partitioning strategies using a constant latency value (like LogP [27]), do not yield the expected high performance. (An example in Chapter 5 illustrates the loss in performance when contentions are accounted for.) A good performance model accounting for the impact of a multithreaded program execution on the system performance is essential for performance related optimizations.

3.2 Problems Studied in this Thesis

In this section, we describe the problems solved in this thesis. The discussion in Section 3.1 shows the necessity to monitor and account for the performance of the processor as well as other subsystems. Therefore our key performance measures of interest are— processor utilization, network latency, and message rate to the network.

Problems in this thesis focus on predicting and characterizing the performance measures using significant architectural and program workload parameters, identifying the system bottlenecks, and providing insights to the performance related optimizations. In later chapters, we develop models of multithreaded systems to predict the performance measures, and apply the models to solve the problems mentioned below.

To a user of multithreaded systems, the processor utilization is the most important performance measure on the effectiveness of all techniques and optimizations in the system. It also provides a uniform measure of effectiveness irrespective of the number of processors in the system. So, our first problem focuses on processor utilization:

Problem 3.2.1 *Given a multithreaded architecture and a program workload:*

1. *How to derive the processor utilization?*
2. *How does the processor utilization vary with model parameters?*
3. *What are the ranges of model parameters which yield high processor utilization?*

The multithreading technique is promoted to be useful to tolerate large latencies on multiprocessor systems. But the multithreading increases contentions. The network perfor-

mance is a critical issue in optimizing an application program, so the network performance is the focus of our second problem:

Problem 3.2.2 *Given a multithreaded architecture and a program workload:*

1. *How to derive the network performance measures- the network latency and the network message rate?*
2. *How does the network performance vary with model parameters?*
3. *What is the relationship between the network performance and the processor performance, with respect to model parameters?*
4. *How robust is the network performance prediction?*

The processor utilization and the network performance, indicate an absolute performance for a set of values of model parameters. Our third problem focuses on how effective is the multithreading to tolerate latencies. Thus, a user knows how much improvement may be achieved through optimizations.

Problem 3.2.3 *Given a multithreaded architecture and a program workload:*

1. *Can we quantify the* latency tolerance*?*
2. *How does the ability of latency tolerance vary with model parameters?*
3. *How is the ability of latency tolerance related to the high processor performance?*

Above three problems address the effect of multithreading on the system performance. Many diverse considerations govern a real system design, e.g. other architectural techniques, an availability of off-the-shelf components, costs of components, and simplicity in the design. So, we perform a case study on McGill's EARTH multithreaded system. The fourth problem summarizes the performance modeling objectives:

Problem 3.2.4 *Given a multithreaded architecture such as the EARTH system and a program workload:*

1. *How do realistic subsystem interactions affect the system performance under a multithreaded program execution? Specifically, how to account for the* simultaneous possession *of the bus in EARTH processing nodes when memory or network interface is accessed.*

2. *How to characterize a realistic multithreaded program workload? That is, how much details should we model about the differences in program workload characteristics at different processing nodes, and service times for different multithreaded operations.*

From a user perspective, a performance model should serve as an aid to the performance related optimizations. Our fifth problem addresses how our performance analysis provides insight to the performance behavior of the EARTH system:

Problem 3.2.5 *Given a multithreaded architecture and a program workload e.g., a loop:*

1. *How to partition the program workload in the presence of realistic long latencies and multithreading costs? What are the significant workload characteristics? What are their critical values to achieve high processor utilization?*

2. *What will be the effect of subsystem implementations on the system performance?*

Next, we outline our approach to solve above problems. In Section 3.4, we describe a multithreaded program execution model which forms a basis to develop analytical models of single processor and multiprocessor systems.

3.3 Our Approach

The problems in Section 3.2 aim to provide a progressively deeper insight to the performance of the multithreading technique. First, we characterize the processor performance, then we study the performance of subsystems, and finally, we analyze real subsystem interactions under multithreaded workload. The above problems are addressed under following aspects– *performance modeling, validation, analysis* and *optimizations.*

Our performance models use closed queueing networks. Initially, we model a single processor multithreaded architecture. This simple system is characterized using basic parameters like the number of threads, their runlengths, the number of memory ports and

the memory access time. An exact solution exists for such a model (which is much simpler than Barrera's method [80]). Then, we analyze a multithreaded multiprocessor architecture. Without loss of generality, we use a 2-dimensional network, a low-bandwidth network, to demonstrate the effectiveness of multithreading technique. We introduce additional program workload and architectural parameters to characterize the multithreaded program execution. Since an exact solution is no longer computationally feasible to analyze a large system, we use approximate mean value analysis (AMVA) [75, 56]. Finally, we model the EARTH system. Architectural related extensions to our performance model include realistic subsystem interactions. We also expand the program workload parameters. We develop heuristics to the MVA to account for above extensions under a multithreaded program workload.

We use discrete-event simulations as well as actual program executions to validate our model predictions. Stochastic timed petri net (STPN) models are developed to verify our predictions for abstract multithreaded systems— a single processor and a multiprocessor system with a 2-dimensional network. These STPN models provide flexibility to simulate various configurations, and require 1 to 5 minutes for each set of parameters on SPARCStation-20 (note that our analytical model predictions require less than 1 minute) On the EARTH system, we execute synthetic and real benchmark programs and measure the performance. The measurements are compared with model predictions for these benchmark programs.

A performance analysis allows us to weigh tradeoffs for a multithreaded program execution. We characterize the performance behavior of a multithreaded system under synthetic program workload. Such an analysis provides an insight to how various program workload characteristics affect the system performance. We show what are the critical values of parameters to avert performance bottlenecks and achieve a high performance. Through examples, we use our characterization to aid performance optimization. Analyses on abstract systems show the best possible gains for multithreaded systems. However, a characterization on the EARTH system shows how large are the realistic multithreading overheads under a multithreaded program workload, and how to reduce these overheads. We also use a case study on real benchmark programs to show the usefulness for performance optimizations.

3.4 A Multithreaded Program Execution Model

To solve the problems listed in Section 3.2, we need a multithreaded program execution model with the following considerations. The model should capture realistic behavior of the program execution on a multithreaded system. Yet, the model should be simple for analytical performance modeling.

Without loss of generality, we describe our multithreaded program execution model assuming an underlying, abstract multiprocessor system with distributed shared memory. We focus on the steady state behavior of multithreaded system necessary for performance modeling (Section 2.2. Such an execution model is easy to adapt to both a single processor system as well as a specific real multiprocessor system.

For the purpose of this thesis, a multithreaded program workload is a collection of partially-ordered threads. Each thread is a sequence of computation instructions followed by a multithreading operation. A multithreading operation involves multiple phases such as accessing the local memory, sending messages on the network, receiving responses, and performing synchronizations. The scheduling of individual threads is similar to a dataflow model. Threads repeatedly undergo the following sequence of states (see Figure 2.3):

Execution: Once scheduled, a thread is executed on the processor pipeline till long latency access is encountered.

Suspension: When the processor issues a long latency operation, and suspends the thread till the response is received.

Ready: A thread becomes ready for execution, after the response to its long latency access is received.

Each processor executes a set of n_t concurrent threads (see the program workload in Figure 2.2). A processor executes a thread for a duration called *runlength* R, before suspending it. On suspension, the state of the outgoing thread is saved and the context of the newly scheduled thread is restored. We assume a fixed context switch time of C cycles. (In practice, C has a variable component depending on how many registers need to be saved.) Threads interact only through long latency accesses. Threads do not migrate across processing nodes.

A long latency memory request is sent to a remote memory module with a probability p_{remote}. The remaining fraction of long latency memory access requests get serviced at the local memory. Note that an assumption of $p_{remote} = 0$ adapts the program execution model to a single processor system. On the other hand, a specific system, like the EARTH, requires a characterization of different types of memory requests, local or remote access, both may be with or without synchronizations.

We do not explicitly consider the presence of an cache in the system for the following reasons. *First*, given the diversity of cache organizations, based on associativity and data sharing, caches introduce too many variables in the model. Further, there is little agreement on how to beneficially utilize the cache in a multithreaded program execution. That's why, multithreaded systems like TERA [9] avoid caches. *Second*, thread runlengths are an embodiment of the cache effect, because a memory access at the end of a thread runlength is same as a cache miss. Some approaches [51, 55] use each cache miss to decide for a context switch, while others [5] use only remote memory cache misses to switch the context.

Above multithreaded program execution model forms a basis for the development of our analytical performance models.

3.5 Summary

This chapter outlined our overall approach to the performance modeling and analysis of multithreaded architectures.

We discussed the issues in the performance modeling of multithreaded architectures, in contrast to single threaded multiprocessor systems and multitasking systems. We described the problems solved in this thesis. These problems are representative of what a user of multithreaded systems will face in practice, e.g., what is the processor performance for a workload, how does the network performance change with various optimizations, and how do the interactions on a real system affect the performance. Then we described our multithreaded program execution model. This execution model forms a basis to develop analytical performance models, and solve the problems on the performance issues, in later chapters.

Chapter 4

Single Processor System

In Chapter 3, we described the performance related problems of interest to this thesis. We also outlined our approach to the performance modeling, validation and analysis of multithreaded architectures.

This chapter focuses on a single processor system with the processor and memory subsystems, specifically the problem 3.2.1 on the processor performance. The two objectives of this chapter are as follows: *First*, develop an analytical performance model of a single processor system; and *second*, analyze the system performance, identify the bottlenecks and suggest optimizations to achieve high processor utilization. A performance behavior of such a single processor system also indicates the maximum achievable performance by a node in a multiprocessor system.

Our results provide interesting insights to the performance of single processor multithreaded architectures. The application parallelism can be exploited as long as the hardware parallelism is not exhausted. That is, the processor utilization increases with the number of threads until all memory ports are busy. Further, the processor utilization is high if the granularity of threads is larger than the *effective memory latency*, defined as an average duration between successive responses from the memory (Section 4.2). For thread granularities closer to the effective memory latency, multithreading provides a high speedup over a single threaded execution.

This chapter is organized as follows. The next section describe the single processor system under discussion. Section 4.2 presents an analytical model of the system and its

exact solution. Section 4.3 provides a characterization of performance measures using architectural and program workload parameters. Section 4.4 summarizes the results of this chapter, and discusses their impact on the system design.

4.1 Architecture

We describe the single processor multithreaded architecture in this section.

Figure 4.1 shows a single processor system with the processor and memory subsystems. The processor subsystem consists of an execution pipeline of the processor and a thread management unit. Register windows may be used to achieve a low context switch time. While executing an application program, the processor uses the data residing in its cache. Note that to ensure a constant number of read/write accesses in a program execution and their impact on the performance due to various cache organization, we ensure that cache misses are explicitly known to us. Henceforth, we refer to these cache misses as the memory accesses. The processor supports multiple outstanding memory accesses. When a cache miss occurs, the processor can continue the execution on the computation, which does not depend on servicing of this cache miss. In other words, on a cache miss, the processor sends a request to the memory and rapidly switches to the computation on another thread. When the memory access is serviced, the corresponding computation thread is ready for the execution.

The memory subsystem supports multiple concurrent accesses to provide a small response time. The contentions at the memory subsystem are reduced using either multiple ports at the memory or interleaved memory banks (numbered 0 to 7 in Figure 4.1). A reduction in the contention reduces the waiting time for each access before it is serviced by the memory subsystem. On servicing the access, the memory subsystem sends the response with the identification of the thread which issued the access.

To support multithreading on a single processor system, the following architectural considerations should be made. Latencies for memory accesses range from 15 to 100 cycles. At conservative processor speeds, memory access times are 15 to 25 cycles, e.g., KSR-1 [19], CM-5, SPARC, MANNA [20, 46]. For aggressive processor speeds, memory access times appear higher, e.g., 60 to 100 cycles in AlphaServer 7000 and 8400 [28]. To exploit the

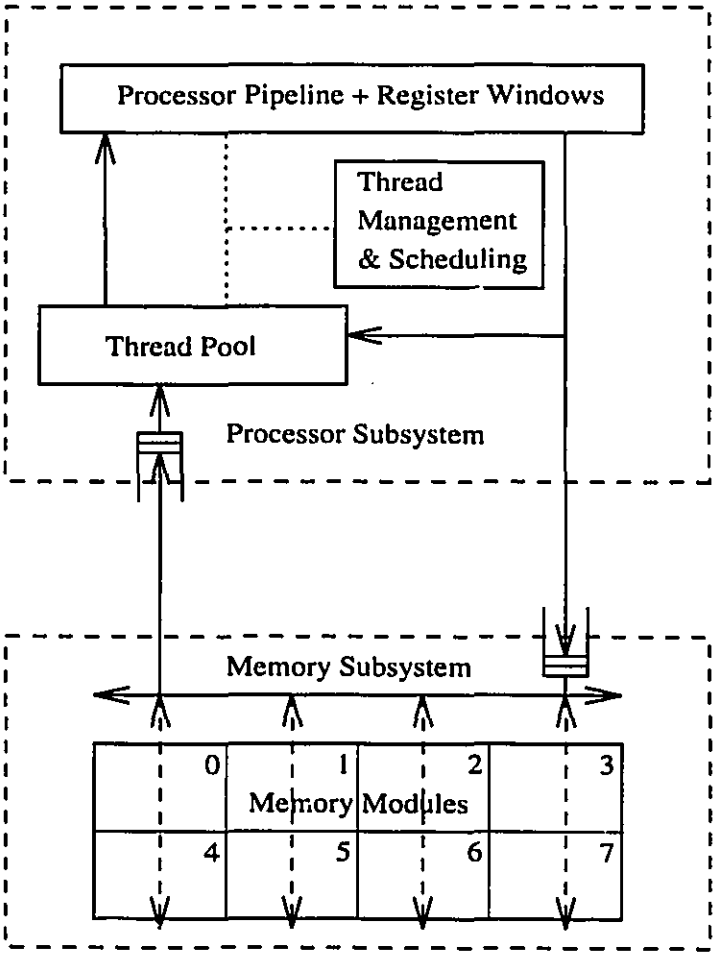


Figure 4.1: A Multithreaded Architecture.

benefits of the multithreading, the context switch overhead should be smaller than the unloaded memory latency. Mechanisms to select a ready thread may differ, e.g., poll for a ready thread, or receive a signal that a thread is ready. Apart from saving the context for one thread, a selection for another thread also incurs an overhead. To gain some performance benefit of multithreading, we should be able to perform two to three context switches in search of a ready computation thread. That is, the overhead of context switch should be less than one-third the unloaded latency.

We consider the following example to illustrate how the performance of a single threaded execution gets limited by the memory system performance. Let the composition of load/store access be 33% of the total instructions in a program, e.g., SPEC workloads have 20 to 50% instructions as load/store operations [28]. Let the cache miss ratios within 5% for large caches [78]. That is, 1.65% (=5% × 33%) instructions are memory accesses. In other words, on average once in 60 instructions a local memory access is required. With one clock per instruction and a memory access time of 20 cycles, the processor utilization is 75% ($= \frac{60}{60+20}$). At aggressive processor speeds, the utilization is significantly low (37.5%= $\frac{60}{60+100}$). With an increase in the use of instruction level parallelism (say a sustained parallelism of 2), computation time on processor decreases (to 30). This further aggravates the memory bottleneck. For a processor supporting multiple outstanding requests, multithreading and prefetching techniques help to alleviate this performance bottleneck.

4.2 Analytical Model

In this section, we will develop an analytical model and its solution to analyze the single processor system in Figure 4.1. We describe our approach using closed queueing networks. Then, we derive the performance measures of interest— processor and memory utilizations.

4.2.1 Closed Queueing Networks

A queueing network is typically useful to model and analyze large systems in their steady state. The main components of a queueing network are the *service centers* and the *customers* visiting these service centers for a service. Many types of distributions are possible for the service time at a service center, e.g., *exponential, deterministic,* and *hyper-exponential.*

The number of customers at each service center represents the *state* of a queueing network. Under steady state, an analysis of the state space of the queueing network yields the performance measures like the utilization of a service center (say, processor), and the response time of a service center (say, the memory).

The exponential service time distribution is most commonly used, due to its *memory less* property. That is, the expected service time of the exponential distribution is always the average value of the service time distribution. We can build *markov* chains using the *memory less* property, where the probability of a transition to another state depends only on the current state and not on the past states the system may have visited. An analysis of markov chains yields the performance results for the queueing network. Some of the good books on queueing networks for the performance analysis of computer systems are by Lazowska *et al* [56], Trivedi [95], and Kleinrock [52].

A mapping of a computer system to a queueing network model is fairly straightforward. Various functional units can be represented as service centers in the queueing network. The hardware delays at these functional units are the service times at service centers. A software program workload determines the characteristics of the service at the processor. Accesses sent by the processor to various functional units represent the customers in a queueing network. With the use of deterministic and exponential distributions, the service time at various service centers can be captured. Figure 4.2(a) shows such a queueing network model of the single processor system in Figure 4.1. `Pr1` represents the processor node, and `R` is the service time for each thread. Servers `S1` to `Snp` represent ports at the memory subsystem, and L is the service time at each port. Arcs connecting the two subsystems maintain queues, `ready pool` holds threads which are ready for execution, and `mem queue` holds outstanding accesses to the memory subsystem.

An advantage of using queueing networks over techniques like Petri Net models, also requiring a state-space analysis, is that sophisticated numerically efficient techniques have been developed to analyze large queueing networks [75, 56]. The benefits become apparent while analyzing a multiprocessor system with a large number of architectural and program workload parameters. To our knowledge, efficient solutions to performance predictions using Petri Net based models are based on the analysis of equivalent queueing networks [96]. We note, however, that Petri Net models present an attractive approach to verify the correctness of system models, and simulate them quickly. Stochastic Petri Nets with certain properties

have been shown equivalent to queueing networks [44, 60]. In the light of this equivalence, we simulate the Petri Net model shown in Figure 4.2(b) to verify our analytical model predictions in this chapter.

4.2.2 The Model and Its Solution

In this section, we develop a performance model for a single processor multi-threaded system in Figure 4.1. The model is a closed queueing network. We show an exact solution to this model for a system with *finite* memory ports. Then, we obtain simple expressions for certain interesting cases, like a *single-port* memory and an *infinite-port* memory. First, we describe the model. Second, we derive the equilibrium probability for the state of the system. Third, we compute the performance measures, like utilizations for processor and memory.

There are two reasons for a detailed description of our simple analytical model based on closed queueing networks (CQN). Saavedra [80] developed a Petri Net model of a similar system and proposed a complex solution by analyzing the state-space. Our CQN model has a very simple solution and yields precisely the same results. Also, unlike Saavedra's solution, the CQN model can be easily extended to model multiprocessor systems and analyzed by modifying the existing techniques (discussed later in Chapters 5 and 7).

We summarize the program execution model, as discussed in Chapter 3 (Section 3.4). A single processor system uses a multithreaded program workload to improve the processor utilization in the presence of local memory latency. We focus on the parallel portions in a program workload, i.e., the workload consists of a number of iterations ($= n_t$) of a `do-all` loop as discussed in Section 2.2. Each iteration is a thread, and it executes for a duration called thread runlength, R cycles, before accessing the memory. Thus, one outstanding request per thread is allowed. When the memory services an access, the computation thread waiting for this access is triggered. Threads interact through memory locations.

A closed queueing network model for the single processor multithreaded system is shown in Figure 4.2(a). Two service centers represent the processor and memory subsystems. The processor executes the workload having n_t threads. The service time R for a thread at the processor is determined by the cache parameters. We note that cache misses are explicitly known. The duration between these misses for a thread is R. On encountering a long latency memory access, the processor sends the access (as a customer) to the memory subsystem,

and suspends the execution on that thread. Assumptions for the closed queueing network model are as follows, and symbols are summarized in Table B.1 of Appendix B:

- The *processor node* is a single server, with a first-come-first-served service discipline. The mean value of the service time for each thread is R cycles. Every thread incurs a context switch time of C cycles at the suspension. Thus, the maximum service rate at the processor, μ_p, is $\frac{1}{R+C}$.

- The *memory node* is a multiple server, with a first-come-first-served discipline at each server. Each of n_p memory ports have one server, and have a service time of L cycles. The overall service rate at the memory, μ_m, is:

$$\begin{aligned} \mu_m &= \frac{xm}{L} \quad , \; xm \leq n_p. \\ &= \frac{n_p}{L} \quad , \; n_p < xm \leq n_t. \end{aligned} \tag{4.1}$$

  where, xm is the number of accesses at the memory, and n_t is the number of threads. μ_m saturates at $\frac{n_p}{L}$, when all memory ports are busy.

- The service times are exponentially distributed.

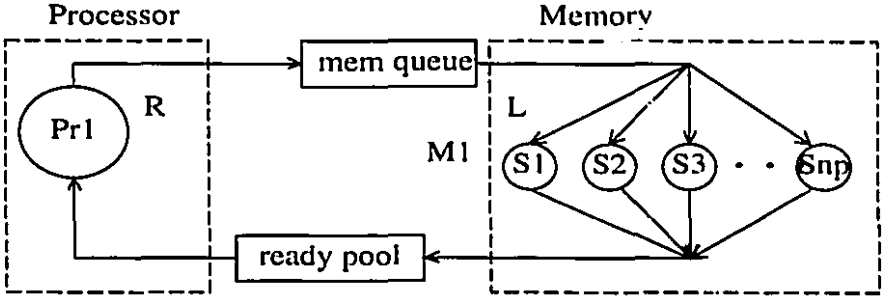
- Processor utilization U_p is the fraction of time the processor executes on threads.

- Memory utilization U_m is the fraction of time for which a memory port is busy (averaged over all memory ports). U_m compares the effectiveness of a memory subsystem with n_p ports to that of a memory subsystem with one port having a service time of $\frac{L}{n_p}$ cycles.

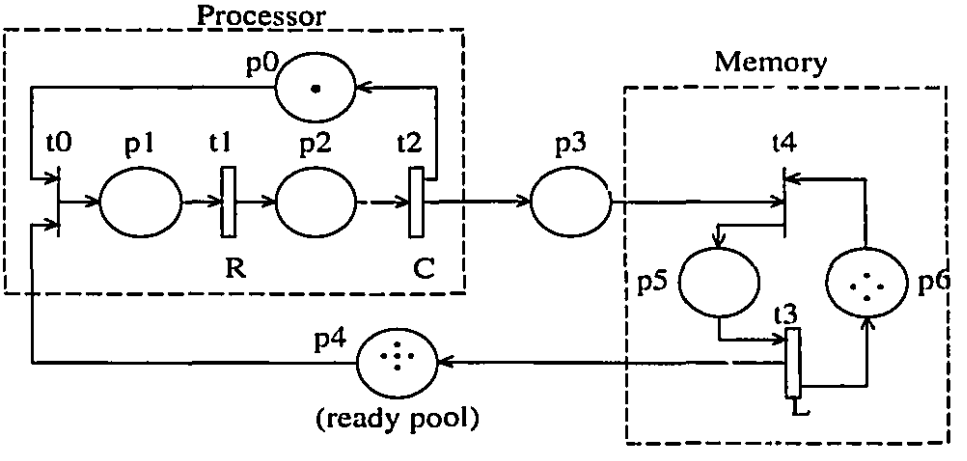
State of the System:

The state of the system, **S**, is defined by the distribution of n_t threads on the two nodes. Let xm be the number of accesses (one per thread) at the memory node, then **S**= $(n_t - xm, xm)$, where $(n_t - xm)$ is the number of threads at the processor node. Thus, xm can used to define the state of the system **S** as shown in Figure 4.3.

We derive the probability of the state **S** as follows. The closed queuing network in Figure 4.2 follows the assumptions in a *product-form* network (see Appendix A and Baskett *et al* [15]). For such a network, the equilibrium probability of a state **S** is given by:



THE QUEUEING NETWORK MODEL



THE PETRI-NET MODEL

Figure 4.2: (a): Queueing Network Model and (b):Stochastic Petri Net Model.

$$P(\mathbf{S}) = G \times fp(n_t - xm) \times fm(xm) \tag{4.2}$$

where, fp and fm are contributions from the processor and the memory module to the equilibrium state probability; and G is a normalizing constant over the state space of $\mathbf{S}$.

Now, we compute the individual terms in Equation 4.2.

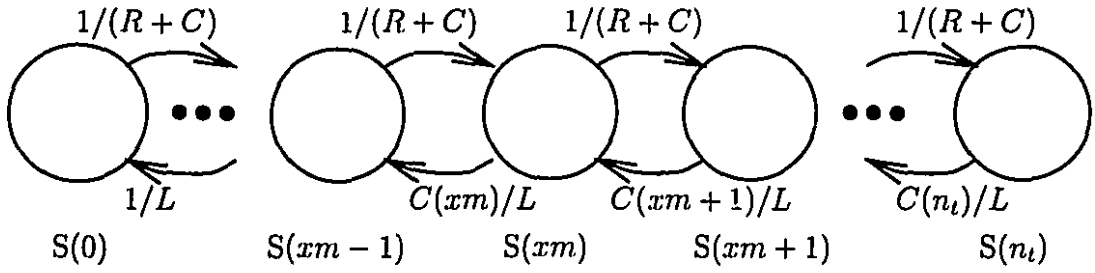


Figure 4.3: State Diagram for a Single Processor Multithreaded System.

The processor node is a single server. In the state $(n_t - xm, xm)$, the number of threads at the processor is $(n_t - xm)$. Since the service rate at the processor remains μ_p irrespective of the number of threads at the processor, $fp(n_t - xm)$ is recursively obtained as follows:

$$\begin{aligned} fp(n_t - xm) &= \left(\tfrac{1}{\mu_p}\right) fp(n_t - xm - 1) \\ &= \left(\tfrac{1}{\mu_p}\right) \prod_{a=1}^{n_t - xm - 1} \left(\tfrac{1}{\mu_p}\right) = \left(\frac{1}{\mu_p}\right)^{(n_t - xm)} \end{aligned} \tag{4.3}$$

The memory node has multiple servers, i.e., n_p ports. In state $(n_t - xm, xm)$, the number of access in service (or waiting for the service) is xm. The service rate is $C(xm)\ \mu_m$, where $C(xm)$ is the number of busy ports. So, $fm(xm)$ can be recursively obtained from $fm(xm - 1)$ as follows:

$$\begin{aligned} fm(xm) &= \frac{\left(\frac{1}{\mu_m}\right)}{C(xm)} \times fm(xm - 1) \\ &= \frac{\left(\frac{1}{\mu_m}\right)}{C(xm)} \times \prod_{a=1}^{xm-1} \frac{\left(\frac{1}{\mu_m}\right)}{C(a)} = \left(\frac{1}{\mu_m}\right)^{xm} \times \frac{1}{\prod_{a=1}^{xm} C(a)} \end{aligned} \tag{4.4}$$

$$\text{where, } C(a) = a \quad \text{for } a \leq n_p$$
$$= n_p \quad \text{otherwise.} \tag{4.5}$$

We substitute the second and third term in Equation 4.2 by their values in Equations 4.3 and 4.4. Below, we rearrange the terms by multiplication with "1" (terms enclosed in circular brackets {}) and redefine the normalizing constant G' (terms in square brackets [.]). Note that normalizing constants G and G' are obtained by summing the equilibrium state probabilities over the whole state space **S**. Thus, we obtain the probability of state **S** as follows:

$$P(\mathbf{S}) = G \left(\frac{1}{\mu_p}\right)^{(n_t - xm)} \frac{\left(\frac{1}{\mu_m}\right)^{xm}}{\prod_{a=1}^{xm} C(a)}$$

$$= G \left(\frac{1}{\mu_p}\right)^{(n_t - xm)} (\mu_m)^{-xm} \left\{ \left(\frac{\mu_m}{\mu_m}\right)^{n_t} \frac{\prod_{a=0}^{n_t} C(n_t - a)}{\prod_{a=0}^{n_t} C(n_t - a)} \right\} \frac{1}{\prod_{a=0}^{xm} C(a)}$$

$$= \left[\frac{G \left(\frac{1}{\mu_m}\right)^{n_t}}{\prod_{a=0}^{n_t} C(n_t - a)} \right] \left(\frac{\mu_m}{\mu_p}\right)^{(n_t - xm)} \prod_{a=0}^{n_t - xm - 1} C(n_t - a) \left\{ \frac{\prod_{a=n_t - xm}^{n_t} C(n_t - a)}{\prod_{a=0}^{xm} C(a)} \right\}$$

$$= \left[\frac{G \left(\frac{1}{\mu_m}\right)^{n_t}}{\prod_{a=0}^{n_t} C(n_t - a)} \right] \left(\frac{\mu_m}{\mu_p}\right)^{(n_t - xm)} \prod_{a=0}^{n_t - xm - 1} C(n_t - a) \tag{4.6}$$

$$= G' \left(\frac{\mu_m}{\mu_p}\right)^{(n_t - xm)} \prod_{a=0}^{n_t - xm - 1} C(n_t - a) = G' \left(\frac{\mu_m}{\mu_p}\right)^{(n_t - xm)} \prod_{a=xm+1}^{n_t} C(a) \tag{4.7}$$

$$1/G = \sum_{\mathbf{S}} \left(\frac{\mu_m}{\mu_p}\right)^{(n_t - xm)} \frac{\left(\frac{1}{\mu_m}\right)^{xm}}{\prod_{a=1}^{xm} C(a)} \tag{4.8}$$

$$1/G' = \sum_{\mathbf{S}} \left(\frac{\mu_m}{\mu_p}\right)^{(n_t - xm)} \prod_{a=0}^{n_t - xm - 1} C(n_t - a) = \sum_{\mathbf{S}} \left(\frac{\mu_m}{\mu_p}\right)^{(n_t - xm)} \prod_{a=xm+1}^{n_t} C(a) \tag{4.9}$$

The above equations represent the **general case** of a single processor system, where the number of memory ports is *finite*. With the steady state probability of the state **S**, we obtain various performance measures. Next, we discuss two special cases of the state space in Figure 4.3, a memory with *infinite* ports, and a memory with a *single* port.

Memory with infinite ports:

When each access to the memory subsystem has a port available for service (i.e. n_p=n_t), there is no queuing delay at the memory. So, from Equation 4.4, $fn(xm)$ simplifies to the

following:

$$fm(xm) = \frac{\left(\frac{1}{\mu_m}\right)^{xm}}{\prod_{a=1}^{xm} C(a)} = \left(\frac{1}{\mu_m}\right)^{xm}\left(\frac{1}{xm\,!}\right) \tag{4.10}$$

We obtain the probability of the state **S** from Equation 4.2 by substituting values from 4.3 and 4.10. We also rearrange the terms similar to that in Equation 4.7.

$$P(\mathbf{S}) = G\left(\frac{1}{\mu_p}\right)^{(n_t-xm)} \frac{\left(\frac{1}{\mu_m}\right)^{xm}}{xm\,!}$$

$$= G'\left(\frac{\mu_m}{\mu_p}\right)^{(n_t-xm)} \times \frac{n_t\,!}{xm\,!} \tag{4.11}$$

$$1/G = \Sigma_{\mathbf{S}}\left(\frac{1}{\mu_p}\right)^{(n_t-xm)} \frac{\left(\frac{1}{\mu_m}\right)^{xm}}{xm\,!} \tag{4.12}$$

$$1/G' = \Sigma_{\mathbf{S}}\left(\frac{\mu_m}{\mu_p}\right)^{(n_t-xm)} \frac{n_t\,!}{xm\,!} \tag{4.13}$$

Memory with a single port:

When $n_p = 1$, the memory node is a single server. The state space for this system is shown in Figure 4.4. Note that the service rate of of memory subsystem is a constant at $\mu_m(=\frac{1}{L})$, that is $C(a) = 1$. So, from Equation 4.4, $fm(xm)$ reduces to the following:

$$fm(xm) = \frac{\left(\frac{1}{\mu_m}\right)^{xm}}{\prod_{a=1}^{xm} C(a)} = \left(\frac{1}{\mu_m}\right)^{xm} \tag{4.14}$$

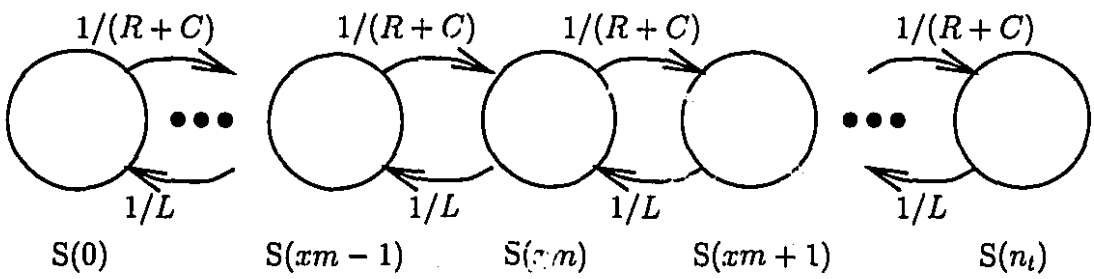


Figure 4.4: State Diagram for a System with Single Memory Port.

Again, we obtain the probability of the state **S** from Equation 4.2 by substituting values from Equations 4.3 and 4.14. Normalizing constants G and G' are derived similar to that

in Equation 4.7.

$$P(\mathbf{S}) = G \left(\frac{1}{\mu_p}\right)^{(n_t - xm)} \left(\frac{1}{\mu_m}\right)^{xm}$$
$$= G' \left(\frac{\mu_m}{\mu_p}\right)^{(n_t - xm)} \quad (4.15)$$

$$1/G = \sum_{\mathbf{S}} \left(\frac{1}{\mu_p}\right)^{(n_t - xm)} \left(\frac{1}{\mu_m}\right)^{xm} \quad (4.16)$$

$$1/G' = \sum_{\mathbf{S}} \left(\frac{\mu_m}{\mu_p}\right)^{(n_t - xm)} \quad (4.17)$$

Performance Measures:

Using the equilibrium probability of the system state **S**, we compute the performance measures of interest, processor utilization, memory utilization, and memory latency.

Processor Utilization:

The utilization of the processor subsystem is the probability that at least one thread is in the processor subsystem. Thus,

$$\text{Utilization of the processor subsystem} = \sum_{xm=0}^{n_t-1} P(n_t - xm, xm) = 1 - P(0, n_t) \quad (4.18)$$

The execution phase of a thread (R cycles) is immediately followed by a context switch time of C cycles. So, U_p is the useful fraction ($\frac{R}{R+C}$) of Equation 4.18.

$$U_p = \left(\frac{R}{R+C}\right) \sum_{xm=0}^{n_t-1} P(n_t - xm, xm) = \left(\frac{R}{R+C}\right) [1 - P(0, n_t)] \quad (4.19)$$

Memory Utilization:

The utilization of the memory subsystem, U_m, is the average utilization of all memory ports. And, the utilization of a memory port is the probability that the port has at least one request for service. So, using Equation 4.5, U_m is defined as follows:

$$U_m = \frac{1}{n_p} \sum_{xm=1}^{n_t} P(n_t - xm, xm) C(xm) \quad (4.20)$$

Memory Latency:

The latency of the memory subsystem is the response time to an access. When the number of ports at the memory is one, we obtain the memory latency L_{obs} as follows (a similar response time equation is derived by Kleinrock [52]):

$$L_{obs} = \frac{n_t/\mu_m}{1 - P(n_t, 0)} - \frac{1}{\lambda} \quad (4.21)$$

where λ is the arrival rate at the memory. Since U_p is the fraction of time the processor is busy, λ is obtained as $\mu_p \times U_p$. When $n_p > 1$, we obtain the average number of accesses $\bar{q}$ waiting in the queue at the memory subsystem. A new access gets service after these $\bar{q}$ accesses are serviced.

$$\bar{q} = \sum_{xm=0}^{n_t} xm P(n_t - xm, xm) \tag{4.22}$$

$$L_{obs} = C(\bar{q})L + (\bar{q} - C(\bar{q}))L \tag{4.23}$$

Now we compute U_p and U_m for three cases based on the number of memory ports:

1. the general case (finite n_p);
2. the ideal case (infinite n_p); and
3. a typical case ($n_p = 1$).

General Case (finite n_p) :

When the number of ports at the memory subsystem is finite (i.e. $n_t \geq n_p$), the transition probability of a state S is given by Equation 4.7. We use Equation 4.19 to obtain the processor utilization.

$$\begin{aligned} U_p &= \frac{R}{R+C} \times (1 - P[\ \mathbf{S}\ (\ 0,\ n_t\ )\ ]) \\ &= \frac{R}{R+C}\left(1 - \frac{1}{\sum_{xm=0}^{n_t}\left(\frac{\mu_m}{\mu_p}\right)^{(n_t - xm)} \prod_{a=0}^{n_t - xm-1} C(n_t\ -\ a)}\right) \end{aligned} \tag{4.24}$$

The denominator in Equation 4.24 is rearranged in terms of $xm \leq n_p$ and $xm > n_p$:

$$U_p = \frac{R}{R+C}\left(1 - 1 \Big/ \left[\sum_{xm=0}^{n_p}\left(\frac{\mu_m}{\mu_p}\right)^{(n_t - xm)} \frac{(n_t)^{n_t - n_p - 1}\, n_p!}{xm!} + \sum_{xm=n_p+1}^{n_t}\left(\frac{\mu_m}{\mu_p}\right)^{n_t - xm} n_p^{n_t - xm}\right]\right) \tag{4.25}$$

We derive the utilization of memory subsystem from Equation 4.20. Substituting the transition probability of a state from Equation 4.7 in square brackets, we obtain:

$$U_m = \frac{1}{n_p} \sum_{xm=1}^{n_t} \left[G \left(\frac{1}{\mu_p} \right)^{(n_t - xm)} \frac{\left(\frac{1}{\mu_m} \right)^{xm}}{\prod_{a=1}^{xm} C(a)} \right] C(xm)$$

$$= \frac{1}{n_p} \frac{[1 - P(xp = 0, xm = n_t)]}{\left(\frac{\mu_m}{\mu_p} \right)} \tag{4.26}$$

$$= \frac{1}{n_p} \frac{R+C}{R} \frac{U_p}{\left(\frac{\mu_m}{\mu_p} \right)} = \frac{1}{n_p} \frac{L}{R} U_p \tag{4.27}$$

Ideal Case (infinite n_p) :

When $n_p \geq n_t$, an access to the memory subsystem does not wait for a service. The transition probability of the state of the system is given by Equation 4.11. The processor and memory utilizations are obtained from Equations 4.19 and 4.20:

$$U_p = \frac{R}{R+C} \left(1 - \frac{1}{\sum_{xm=0}^{n_t} \left(\frac{\mu_m}{\mu_p} \right)^{(n_t - xm)} \left(\frac{n_t !}{xm !} \right)} \right) \tag{4.28}$$

$$U_m = \frac{1}{n_t} \sum_{xm=1}^{n_t} G \left(\frac{\mu_m}{\mu_p} \right)^{n_t - xm} \frac{n_t !}{xm !} xm$$

$$= \frac{1}{n_t} \frac{\mu_p}{\mu_m} \left(\frac{R+C}{R} \right) U_p = \frac{1}{n_t} \frac{L}{R} U_p \tag{4.29}$$

We note a similarity between Equations 4.25 and 4.28. When $xm \leq n_p$, the memory subsystem gives an illusion of being *ideal* in that the queueing delay is *zero*. In the general case described above, the first term in the denominator of Equation 4.25 captures this behavior. Substituting $n_p = n_t$ in Equation 4.25 yields Equation 4.28. The impact of the hardware parallelism (n_p) and the exploited application parallelism (n_t) on the processor performance is similar. As we will see in Section4.3, the second term in the denominator of Equation 4.25 contributes very little to the processor performance (this occurs when $xm \geq n_p$, i.e. hardware parallelism at the memory subsystem is fully exploited). Thus, a matching of hardware parallelism to the application parallelism is necessary to fully exploit the performance gains due to multithreading.

Saavedra [80] and Alkalaj [8] use such an *ideal* case for their analyses. They assume that a multithreaded system provides a fixed latency to access its resources. When a large number of threads are present, the processor utilization will eventually becomes high (note that the denominator in Equation 4.28 becomes very large). Thus, a large latency can be tolerated. This case ignores the contention at the memory, hence is not true in practice. The general case (with finite $n_p < n_t$) and the typical case ($n_p = 1$) are mainly observed.

Typical Case ($n_p = 1$) :

When $n_p = 1$, $C(xm)$ is 1. Thus, Equation 4.19 and 4.20 reduce to the following:

$$U_p = \frac{R}{R+C}\left(1 - \frac{1}{\sum_{xm=0}^{n_t}\left(\frac{\mu_m}{\mu_p}\right)^{(n_t - xm)}}\right) \tag{4.30}$$

$$U_m = \frac{R+C}{R}\frac{U_p}{(\mu_m/\mu_p)} = \frac{L}{R}U_p \tag{4.31}$$

Since the memory subsystem is a single server, the system is symmetric and simple. The maximum achievable processor utilization is defined by the ratio of thread execution time at the processor and the service time for access at the memory. If $R > L$ then processor reaches saturation with large number fo threads, otherwise memory subsystem saturates. Thus, U_p in following two cases are obtained as follows:

1. when $\frac{R}{L} < 1$:

$$U_p = \frac{R}{R+C}\left(1 - 1/\left(\sum_{xm=0}^{n_t \to \infty}\left(\frac{\mu_m}{\mu_p}\right)^{n_t - xm}\right)\right) = \frac{R}{R+C}\left(1 - (1 - \frac{\mu_m}{\mu_p})\right) = \frac{R}{L} \tag{4.32}$$

2. $\frac{R}{L} \geq 1$:

$$U_p = \frac{R}{R+C}\left(1 - 1/\left(\sum_{xm=0}^{n_t \to \infty}\left(\frac{\mu_m}{\mu_p}\right)^{n_t - xm}\right)\right) = \frac{R}{R+C} \tag{4.33}$$

Effective Memory Latency:

In all three cases, U_m is related to U_p through L, R and the maximum number of ports in use (n_p for *general case*, and n_t for *ideal* case). The memory utilization from Equations 4.27, 4.29 and 4.31 is:

$$U_m = \frac{1}{min\{n_p, n_t\}} \times \frac{L}{R}U_p = \frac{L}{min\{n_p, n_t\}} \times \frac{1}{R}U_p \tag{4.34}$$

Intuitively, the maximum number of accesses a memory can service simultaneously is the minimum of its number of ports and the number of threads in the system. So, the service rate at the memory subsystem is $min\{n_p, n_t\}/\mu_m = \frac{min\{n_p, n_t\}}{L}$. We define the *effective memory latency*, L_{eff}, as the reciprocal of the effective service rate at the memory. L_{eff} is on average the minimum time between two successive service completions at the memory subsystem. As shown later, L_{eff} significantly affects the processor performance.

We apply the above analytical model to analyze the performance behavior of the single processor system in Section 4.3.

4.3 Results

In this section, we highlight our results when our model is applied to analyze the performance of a single processor system. Our performance measures of interest are, the processor utilization, memory utilization, and memory latency. We characterize the variation in these measures with architectural and program workload parameters:

- We show that an increase in U_p with an increase in the number of threads is almost to the same extent as the increase in U_p with an increase in the number of threads. A duality exists between n_t and n_p, and a minimum of the two values dominates U_p.
- We show how the processor and memory utilizations vary with the thread runlength and effective memory latency. We also show how to capture the transition region of U_p and U_m curves using system utilization, an average of processor and memory utilizations. We also note variations in memory latency, L_{obs}, with model parameters.
- We show the critical values for architectural and program workload parameters to achieve high processor performance. Specifically, the thread runlengths should match (and exceed) the effective memory latency values to yield high U_p.

Our results are arranged as follows. Section 4.3.1 presents a verification of the model predictions using simulations of Petri Nets. Section 4.3.2 shows the effect of duality of n_t and n_p on the processor performance. Section 4.3.3 shows how R and L_{eff} affect subsystem utilizations. Finally, Section 4.3.4 investigates critical values for parameters to yield a high processor utilization.

4.3.1 Verification of the Model Predictions

Now, we verify predictions of the analytical model presented in Section 4.2. We compare the model predictions with simulation results of the Stochastic Timed Petri Net (STPN) model of the single processor multithreaded system shown in Figure 4.1.

Our choice of STPN model to verify the model predictions is based on the following reasons:

- The STPN models provide an easy approach to quickly modify system designs, and verify their correctness.
- The STPN models with product form solutions are equivalent to the queueing network models [96]. Hence, we can verify the analytical predictions of the queueing network models easily. Moreover, the STPN models provide a flexibility in the use of arbitrary distributions for parameters. For example, we can simulate an actual program execution with the Voltaire package [72], which is used for our purposes. Many similar packages exist. [1]

The STPN model for the single processor multithreaded system under investigation is shown in Figure 4.2(b). The processor and memory subsystems are marked by dotted lines.

The STPN model is a bipartite graph of *place* and *transitions*. A transition receives *tokens* from one or more input places, and delivers them to one or more *output* places. Circles denote *places* which hold *tokens*. Either a token occupies some space in the *places* or keeps a *transition* busy. Voltaire [72] allows various attributes to tokens, which facilitates the processing of tokens. Lines without arrows denote immediate transitions, e.g. $t0$ and $t4$. These transitions fire as soon as their input places have tokens. Rectangular boxes denote transitions with *non-zero* firing times.

The processor subsystem consists of places $p0$, $p1$, and $p2$. A pool of *ready* threads is maintained at the place $p4$. A single token in the place $p0$ ensures that at a time only one thread is executed by transitions $t0$, $t1$, and $t2$. The transition $t1$ executes a thread for the duration R, before a memory access is requested. The memory access is queued at the place $p3$.

[1] Murata [62] provides a detailed survey on Petri Nets. Trivedi et al. [96] discuss some of the recent tools for performance evaluation based on queueing networks and petri nets.

The memory subsystem consists of places $p5$ and $p6$. The number of tokens in $p6$ indicates the number of ports at the memory. For each memory access at $p3$, a port is chosen from place $p6$, and is kept busy for a duration L at the transition $t3$. On service of the memory access, the corresponding thread is placed in the ready pool ($p4$). For our simulations, the duration R is exponentially distributed. L and C are fixed time delays. The analytical model assumes exponential distributions for all non-zero delays.

We perform simulations for 100,000 time units (which is long enough to eliminate the transient effects in simulations). Our performance measure of interest is U_p. We report simulation results and analytical predictions for the following two sets of parameters:

1. *The Ideal Model:* $R = 15$, $C = 2$, $L = 100$, and $n_p = 20$.
2. *The General Model:* $R = 15$, $C = 2$, $L = 100$, and $n_p = 5$.

In Figure 4.5, U_p is plotted with number of threads, for two values of n_p. "Simulation" represents the mean value of U_p obtained from multiple simulation runs. "Model" represents the prediction by analytical model (Equation 4.28 for the *ideal* case and Equation 4.24 for the *general* case). "Asymptote" represents a simple, deterministic analytical model, which we proposed in [65]. The deterministic model is as follows:

$$U_p = \frac{R}{L+R} \quad \text{when } n_t = 1. \tag{4.35}$$

$$U_p = \frac{R \times n_t}{L+R+C} \quad \text{when } n_t \leq n_p \text{ and } R \times (n_t - 1) < L. \tag{4.36}$$

$$U_p = \frac{R \times n_p}{L} \quad \text{when } n_t > n_p \text{ and } R \times n_p < L. \tag{4.37}$$

$$U_p = \frac{R}{R+C} \quad [R \times min\{n_t - 1, n_p\}] \geq L \text{ and } n_t > 1. \tag{4.38}$$

From Equations 4.36 and 4.37, we observe that $n_t \geq n_p$ yields no performance gain.

Ideal Model: $n_p = \infty$

When the number of ports is large (20 in this case), there is almost no queueing at the memory, and is defined earlier as the *ideal* case. Figure 4.5 shows that analytical predictions match well the simulation results (within $\approx 3\%$). A possible source of discrepancy is that the STPN model uses a fixed value of context switch overhead C, while the analytical model uses an exponential distribution.

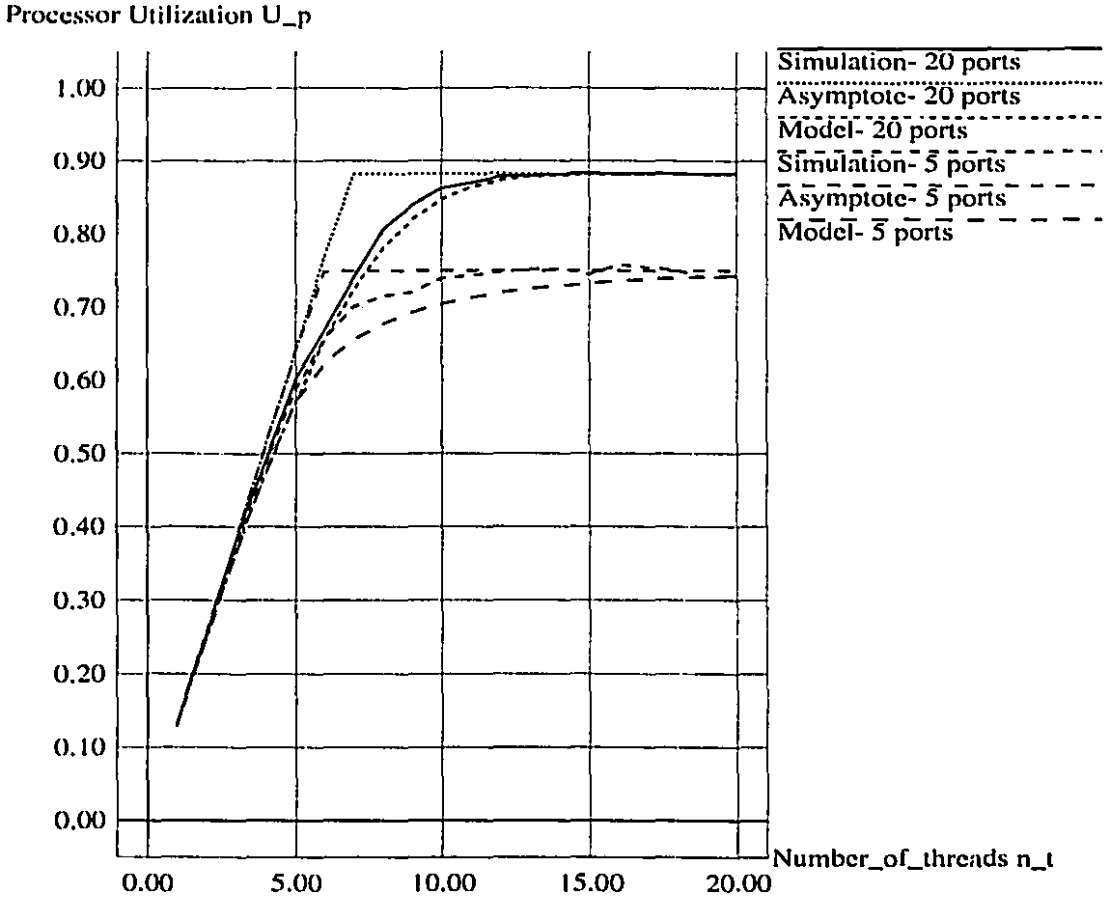


Figure 4.5: Comparing the model with simulation results and the asymptotic analysis: for $n_p = 5, 20$, $R = 15$, $C = 2$, $L = 100$.

These results also match with the analytical results of a more complicated solution of Petri Net model presented by Saavedra-Barrera *et al* [80]. The STPN model obtains a closed form expression for processor utilization [80] by assuming the runlength as a random variable, and keeping other parameters fixed. Such model cannot account for contentions, e.g. the next case shows $n_p \leq n_t$ and contentions occur at the memory.

General Model: $n_p < n_t$

Now, we reduce the number of ports to 5, i.e. less than the maximum number of threads. Analytical predictions by Equation 4.24 match within 5% of the simulation results. Again, a small deviation from the simulation results occurs, which can be attributed to the exponential distribution of C in the analysis, as opposed to a fixed value in the STPN simulations.

In both the *ideal* case and the *general* case, we note that the processor utilization increases with number of threads. The general case also shows that processor utilization saturates, when n_t exceeds n_p. This lends credence to the above simple, deterministic model described by Equations (4.35) to (4.38).

Thus, the performance prediction by the analytical model developed in Section 4.1 compares well with the performance results of STPN simulations. While we report the predictions of analytical model in rest of this chapter, these have been verified using simulations.

4.3.2 Processor Utilization

In this section, we characterize the variation in the processor utilization with architectural and workload parameters. We use the *general* case to derive processor utilization as described in Equation 4.25. When $n_p = 1$, Equation 4.25 converges to Equations 4.30. Similarly when $n_p \geq n_t$, Equation 4.25 converges to Equation 4.28, the *ideal* case.

Figure 4.6 shows how processor utilization changes with the number of threads. To show how the hardware parallelism at the memory subsystem affects the performance, we use a large L value of 100 cycles, and small R of 15 cycles. C is 2 cycles. When the number of memory ports is high, U_p rises rapidly with the number of threads. The saturation value of U_p is high, if n_p is high. For example, at $n_p = 7$, U_p rises to 75% before saturating. On the

other hand, at a low n_p of 4, U_p saturates at a low value 47.5% even when n_t is increased to 10. At low n_p, since L is significantly larger than R, memory services the accesses at a slower rate than that of processor requesting them. So, the processor cannot overlap the memory latency with computations, and the processor utilization is low. In summary, the processor utilization increases with the number of threads when the memory subsystem can service multiple requests simultaneously.

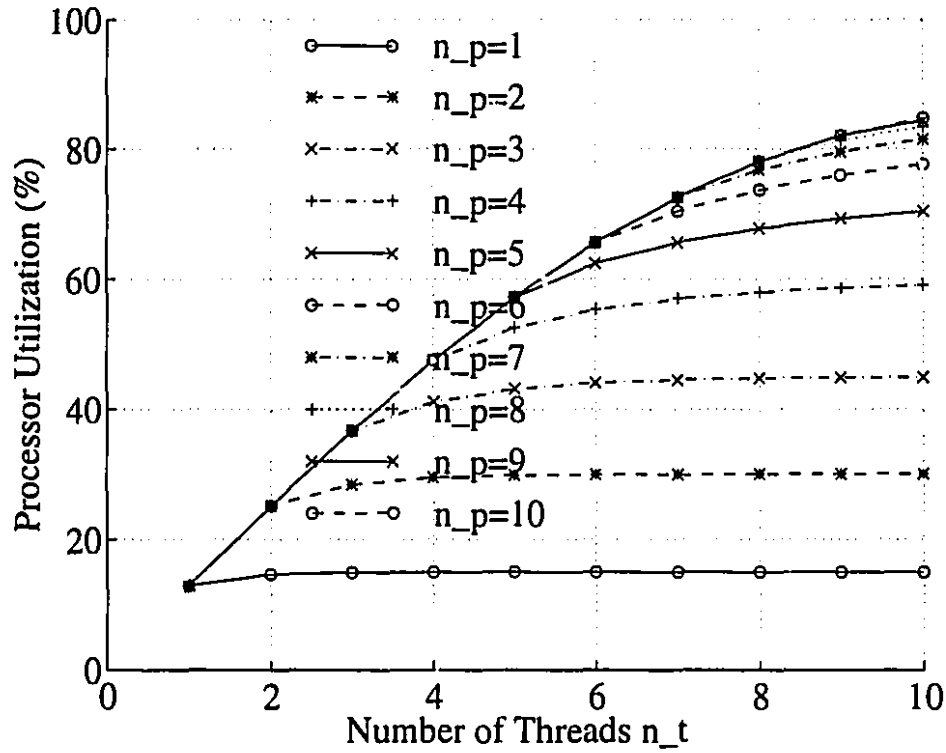


Figure 4.6: Effect of n_t on U_p when $R = 15$, $C = 2$, $L = 100$.

Now, we show how the variation in n_t as well as n_p affects the processor utilization (see Figure 4.7). We consider an average runlength R of 25 cycles. Other parameters are: $C = 2$, and $L = 100$. Interestingly, increasing either of the parameters, n_t and n_p, increases U_p to almost similar values. The lower of the two parameters dominates U_p values. This duality is a result of the available hardware parallelism in the system, e.g., n_p ports at the memory. (Recall the discussion following Equation 4.28.) Note that

U_p saturates for $n_t \geq (n_p + 1)$. Thus, the latency of an *ideal* memory subsystem can be tolerated using $5(= 1 + L/R)$ threads. Figure 4.7 shows that U_p reaches nearly 80%. Intuitively, when n_t equals $n_p + 1$, all n_p memory ports are busy serving memory accesses, and the $(n_p + 1)$-th thread executes on the processor. The surface for U_p values is almost symmetric around $n_t = n_p$. Thus, the parallelism in the program workload improves the performance as long as the parallelism exists among the hardware resources. A use of split-phase transactions allows the multithreading to exploit the concurrency among hardware resources, e.g. simultaneous accesses to multiple memory ports.

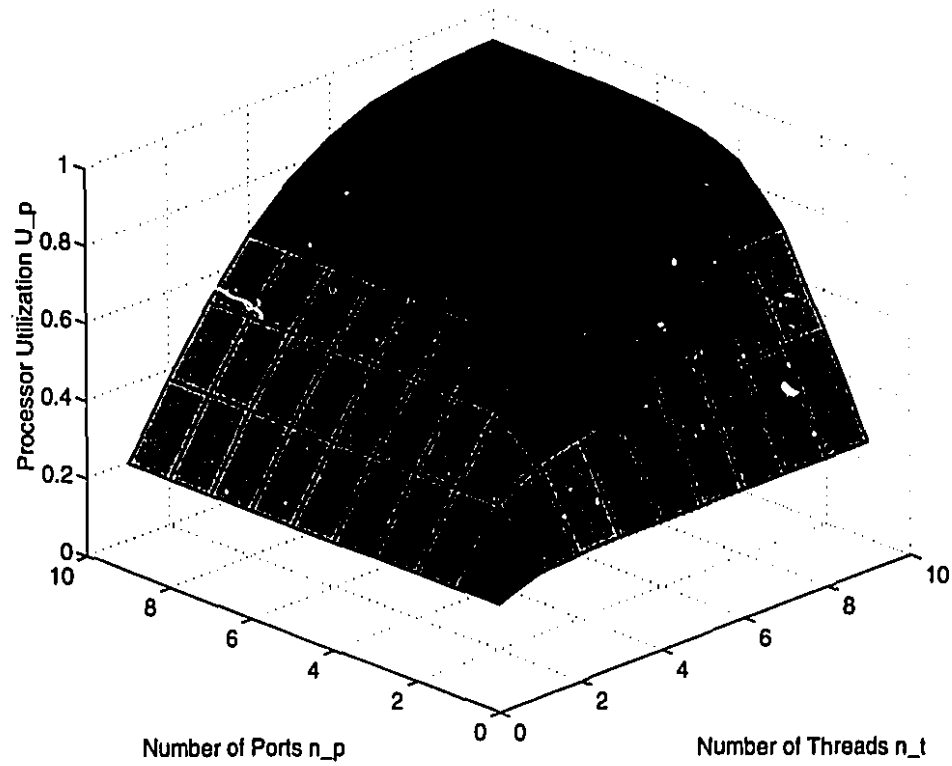


Figure 4.7: Effects of n_t and n_p on U_p, when $C = 2$, $L = 100$, and $R = 25$.

In summary, we note that:

- The parallelism in the program improves the performance as long as the parallelism (concurrency) exists among the hardware resources in the system. A duality exists between n_t and n_p such that same changes in either parameter affects the processor performance by almost the same value.

- The minimum value of the two parameters, n_t and n_p, dominates U_p value. So, the most of the performance gains can be obtained even with a low value of n_t, if $n_t \geq (n_p + 1)$.

4.3.3 Subsystem Utilizations

In this section, we will study how the performance of both subsystems varies with architectural and program workload parameters. Specifically, we track the knee of the processor performance, i.e. transition region where the performance changes are significant.

The effect of thread runlength on utilizations of processor and memory subsystems is shown in Figure 4.8. The processor utilization is denoted by U_p and memory utilization by U_m. We fix the L value as 50, the maximum value of R in the Figure 4.8. We assume that $n_t = 5$, so we fix n_p at 3 to exploit full hardware parallelism. With increasing R values, U_p increases rapidly. Since $n_p = 3$ and $L = 50$, the knee occurs when $R = 17(= \frac{L}{n_p})$. Above this R, U_p values are high.

The variation in U_m with increasing R shows that each thread spends a smaller fraction $\left(\frac{L}{R+C+L}\right)$ at the memory subsystem, so U_m decreases. U_m values start decreasing when R approaches L_{eff} $(= \frac{L}{n_p}$ in this case). The workload is equally balanced between the two subsystems at $R = L_{eff}$, and U_p equals U_m.

For a single processor system, we define the *system utilization*, U_{sys}, as an average of the processor utilization and the memory utilization:

$$U_{sys} = \frac{U_p + U_m}{2} \tag{4.39}$$

We define U_{sys} with the two purposes. *First*, for a high performance, U_p should be high. For a cost-effective performance, we want both U_p and U_m to be high, i.e. U_{sys} should be high. *Second*, we want that U_{sys} should track the transition region, where one subsystem reaches saturation with changes in a parameter while the other subsystem comes out of saturation. U_{sys} is high near the knee of the U_p curve. We call a maximum value of U_{sys} as the *Peak System Utilization*, (*PSU*). The PSU tracks the onset of workload parameter values yielding a high U_p. From Figure 4.8, with increasing R values, the PSU occurs when U_p is near saturation ($\approx 85\%$). Also, the U_p value is close to its maximum.

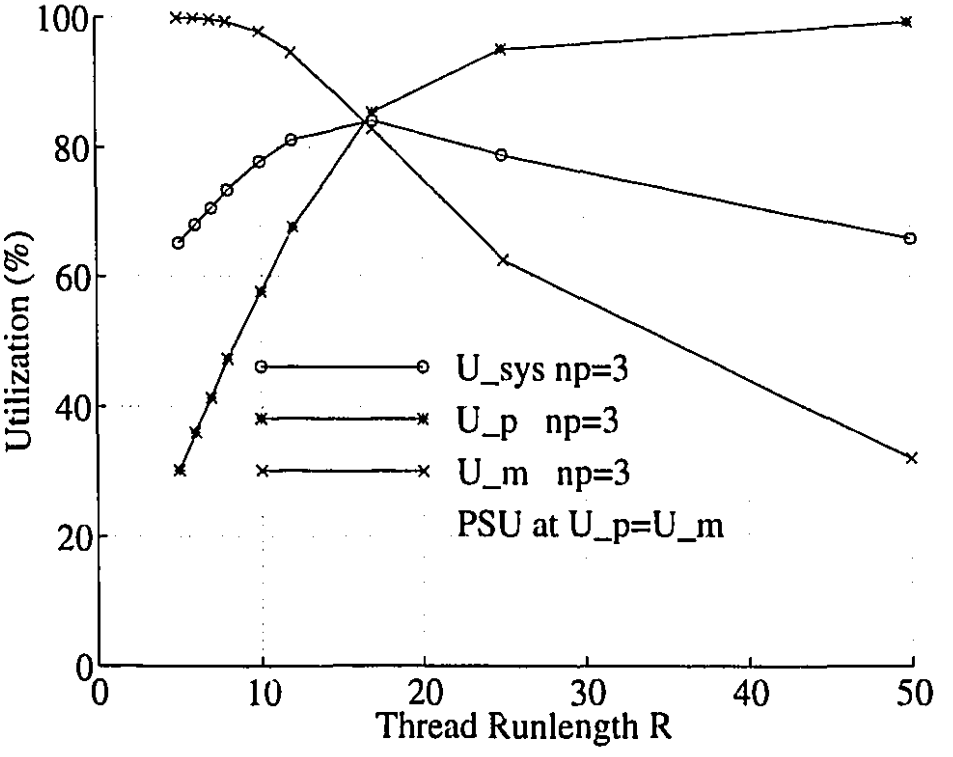


Figure 4.8: U_p, U_m and U_{sys}, when $C = 0$, $L = 50$, and $n_t = 5$.

Figure 4.9 shows a perspective on how the performance measures will change with L_{eff}. U_{sys} behavior with L_{eff} $(= \frac{L}{n_p})$ is similar to U_{sys} behavior with R as seen earlier in Figure 4.8. We note a significant increase in performance of a multithreaded system when concurrent requests at the memory are serviced. U_m and L_{obs} values indicate that queueing delays at the memory are sufficiently low even with 2 to 4 memory ports. For $n_p = 1$, L_{obs} is high. At the PSU i.e. for $n_p = 5$, however, the value of L_{obs} is less than twice the no-load value.

Now, we study the variation of U_{sys} with thread runlength. Figure 4.11 shows how the PSU changes at different L_{eff} values. Other parameters are: $C = 0$, $L = 50$, and $n_t = 5$. We achieve the variation in L_{eff} by changing the number of memory ports. As expected, when L_{eff} is large, the PSU occurs at a large R $(=L_{eff})$. An increase in R also results in a high U_p. So, the PSU $(= U_p = U_m)$ is also high.

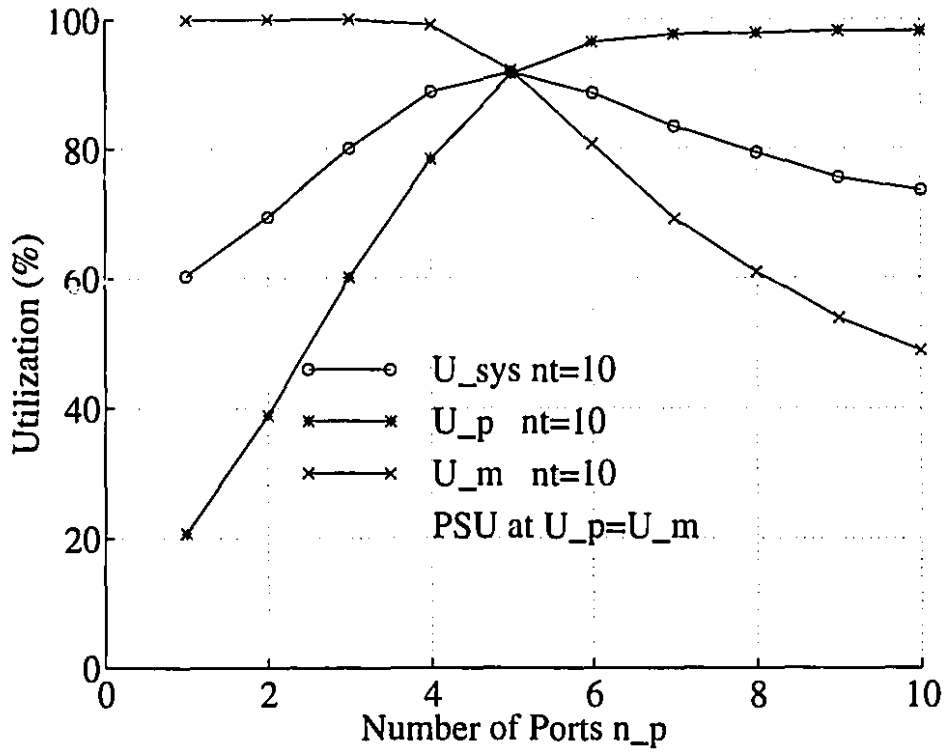


Figure 4.9: Effect of n_p on U_p, U_m and U_{sys}, when $R = 10$, $C = 0$, $L = 50$, and $n_t = 10$.

We show the variation of U_{sys} with L_{eff} in Figure 4.12. U_{sys} curves for R= 12, 25 and 50 cycles, are similar to those in Figure 4.11. We also observe that U_{sys} values increase when the PSU occurs at high R values. Other than service times, the only architecture parameter being changed is n_p. Let us analyze the two equations for processor utilization, Equations 4.28 and 4.30. These equations indicate that n_p memory ports with a mean service time of L cycles do not exactly behave as *one* memory port with a mean service time of $\frac{L}{n_p}$ cycles.[2] However, these results on U_p and U_{sys} for multithreaded systems show that their behavior is indeed similar. We will discuss the design implications of this observation in Section 4.4.

[2]To be precise, each term in the denominator of Equation 4.28 is smaller by a factor $\frac{1}{n_p^{(n_t - xm)}} \frac{n_t!}{xm!}$ with respect to the corresponding term in Equation 4.30. So, increasing L and n_p, to keep $\frac{L}{n_p}$ the same, yields a slightly lower value of U_p.

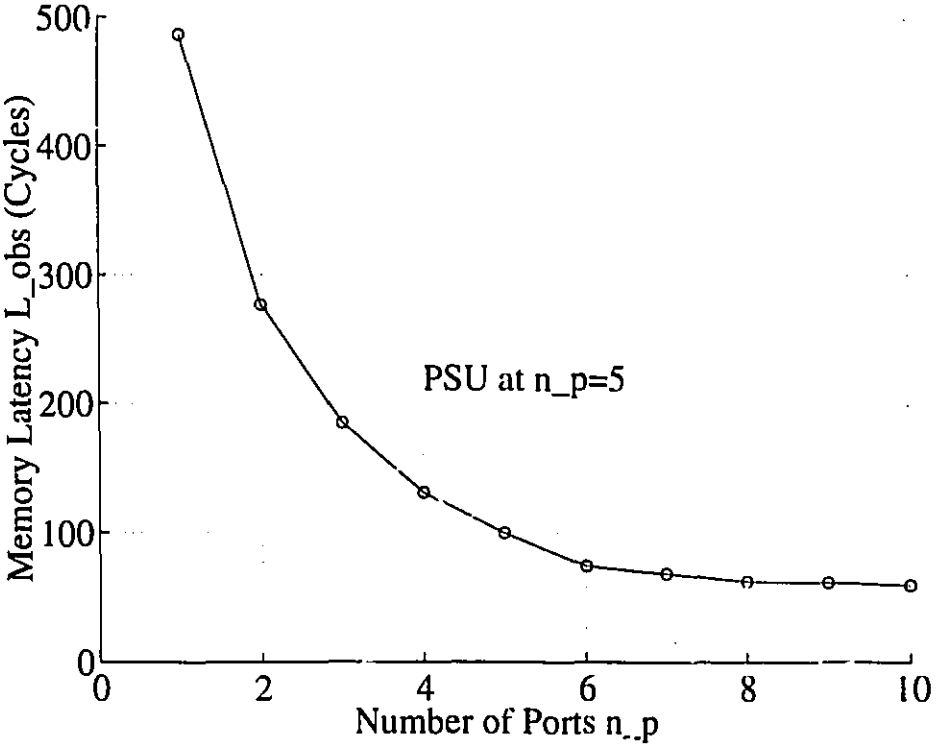


Figure 4.10: Effect of n_p on L_{obs}, when $R = 10$, $C = 0$, $L = 50$, and $n_t = 10$.

In summary:

- System utilization, U_{sys}, is an average of U_p and U_m values. The peak system utilization, PSU, tracks the knee of the processor performance. The PSU occurs, when R equals L_{eff}.
- With n_p ports, the memory subsystem behaves similar to a server with a mean service time of $\frac{L}{n_p} (= L_{eff})$ cycles, under a multithreaded program execution.

4.3.4 Critical Values for the Parameters

Previous sections (4.3.2 and 4.3.3) showed a duality between the processor and memory subsystems. In this section, we study how this duality helps to achieve high processor

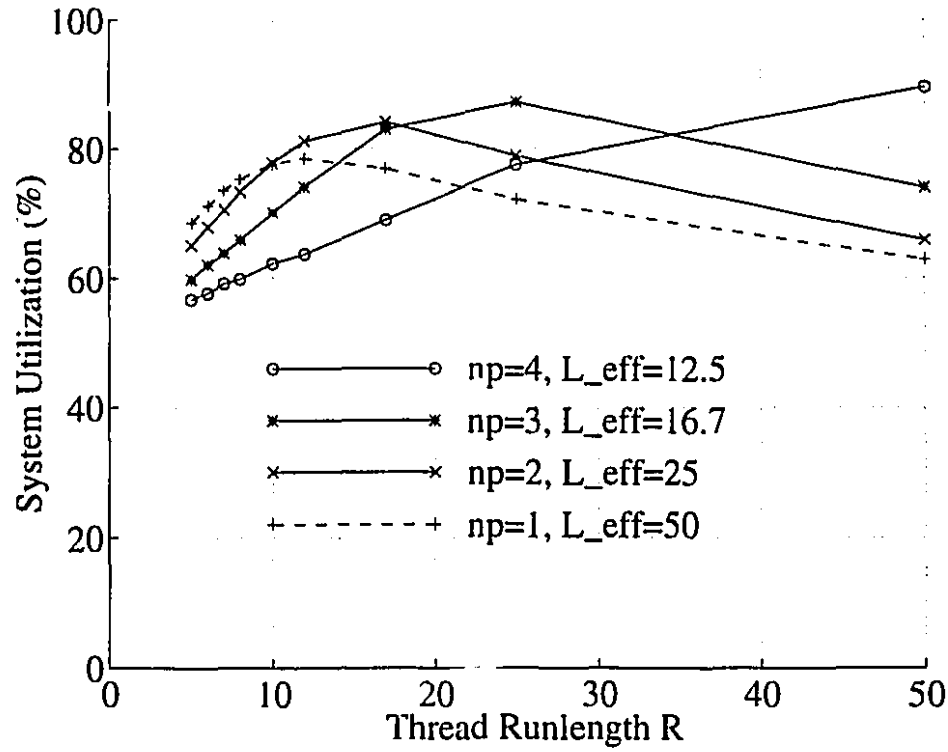


Figure 4.11: Effect of Thread Runlength on System Utilization, when $n_t = 5$, $C = 0$, $L = 50$.

performance. In particular, what the critical values are for program workload parameters—the number of threads and their runlengths.

We have observed that increasing R beyond L_{eff} marks the onset of high U_p region. Figure 4.14 shows the variation in U_p with thread runlength, for different values of L_{eff}. We change L_{eff} by varying the number of memory ports. Other parameters are: $n_t = 10$, $C = 2$, and $L = 100$. In general, $R \geq L_{eff}$ yields a high U_p. However, if R is large when R equals L_{eff}, U_p values is high. For example, when $R = L_{eff} = 100$ and $n_p = 1$, U_p is 90%. When $R = L_{eff} = 10$ and $n_p = 10$, U_p is 72%. (Using $R = L_{eff} = 10$ and $n_p = 1$ in the Equation 4.25 described in Section 4.2, we get $U_p = 81\%$.) In other words, a memory subsystem with a high L and n_p approaches (but does not equal) the performance of a faster memory subsystem, i.e., lower L and a low n_p. The difference is prominent at low R

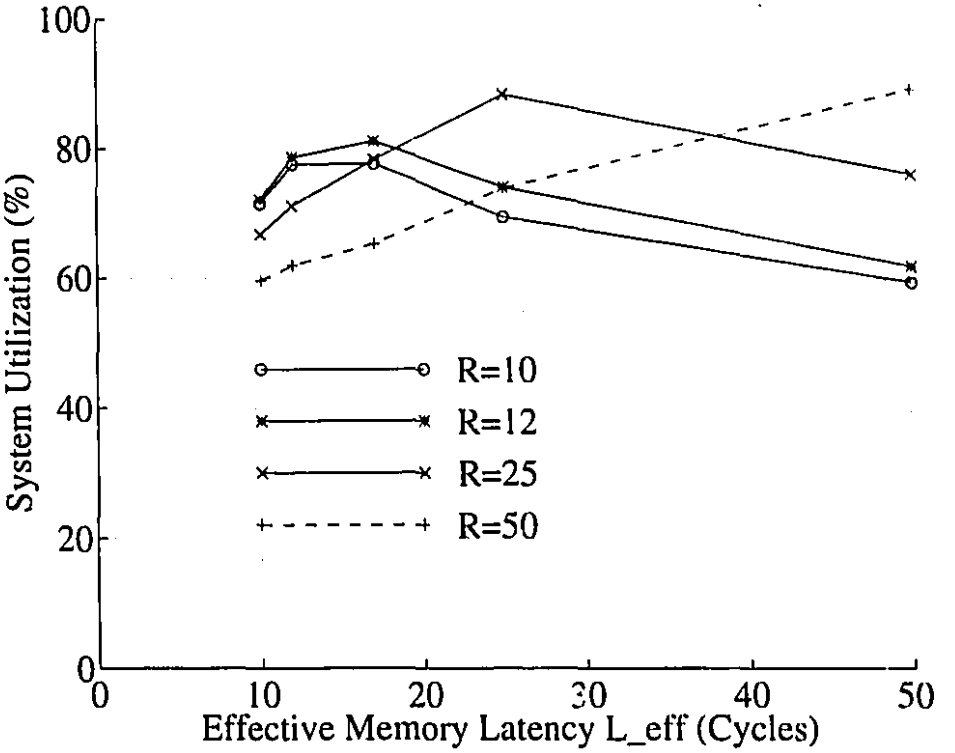


Figure 4.12: Effects of n_p and R on System Utilization, when $C = 0$, $L = 50$, $n_t = 5$.

values. This gives credence to the belief that both low latency and high bandwidth are a requirement for high performance computing [17].

Let us look at the multithreaded program workloads used in practice. Two contrasting types of multithreaded program workload are used in practice: *First,* Schauser et al [85], and Roh et al [77] report thread runlengths of the order of 3 to 30 instructions. *Second,* Maqueline et al [59], and Thekkath and Eggers [90] report thread runlengths of the order of 700 to 1,000,000 cycles. So, *which workload characteristics are suitable for single processor multithreaded system?*

We consider the first group of multithreaded workload, which exhibits a fine granularity of threads, i.e. 3 to 30 instructions. To achieve the condition $R = L_{eff}$, the effective memory latency for an architecture should be low. Increasing the number of memory ports helps to reduce L_{eff}, and achieve high performance on such a program workload. Figure 4.14

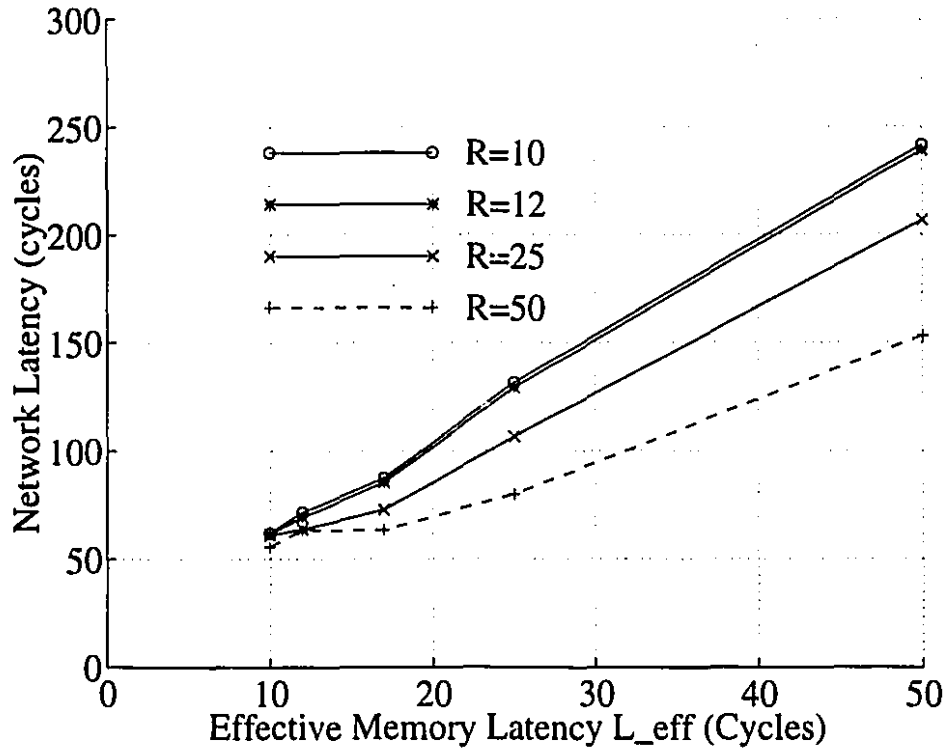


Figure 4.13: Effects of n_p and R on L_{obs} , when $C = 0$, $L = 50$, $n_t = 5$.

shows that 4 to 6 ports are sufficient to provide good performance. Additionally, the use of interleaved memory banks can increase the concurrency among multiple memory requests. An advantage of the interleaving is that the number of interleaved banks can be as high as 16 [76]. Now, consider the second group of multithreaded workload with thread runlengths of the order of 700 to 1,000,000 cycles [59]. Threads do not switch the context for local memory accesses. The thread switching itself requires restoring the thread identifiers from the local memory. Such partitioning is not intended to take advantage of the multithreading technique on a single processor system.

4.4 Summary

The major results for this single processor system are summarized below:

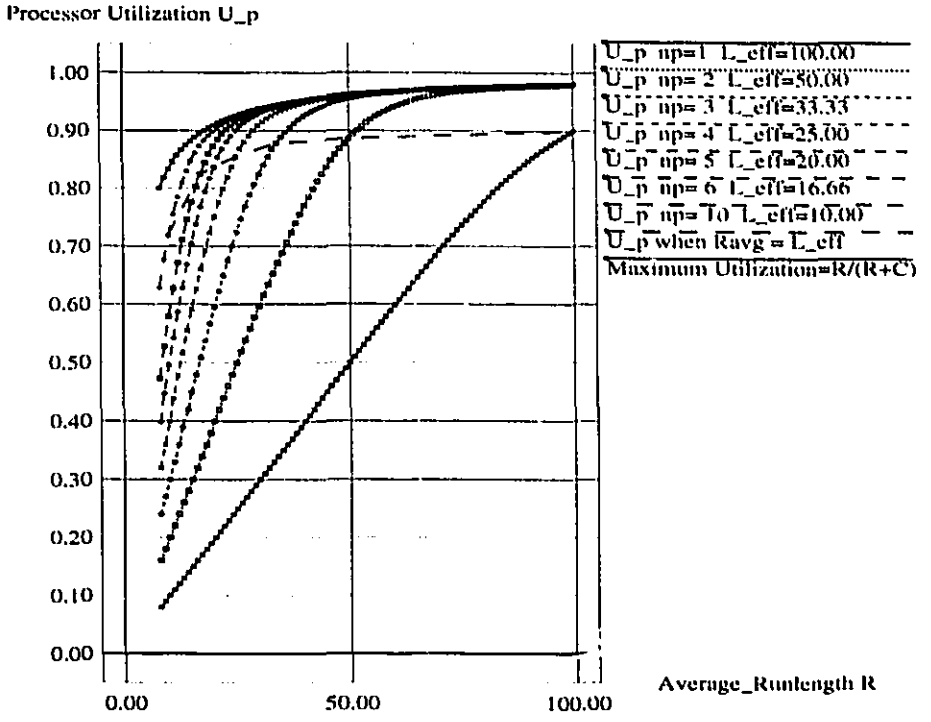


Figure 4.14: Critical values of R for $n_t = 10$, $C = 2$, $L = 100$.

1. The parallelism in a program workload improves the performance as long as the concurrency exists among hardware resources in the system. The result is a duality between processor and memory subsystems. A change in either of n_t and n_p affects U_p by almost the same value. However the parameter, which has a lower value, dominates U_p value.

2. Under a multithreaded program execution, the performance of a memory subsystem, with n_p ports and a latency L, approaches that of the subsystem with 1 port and latency $\frac{L}{n_p} (= L_{eff})$.

3. For $R \geq L_{eff}$, a high U_p is obtained. At $R \approx L_{eff}$, U_p is almost 90% of its maximum value. Corresponding values of L_{obs} are also low.

The above observations have a significant impact on the design of a multithreaded system. Given typical local memory latencies of 20 to 100 cycles [28, 19, 46], a multithreaded program workload should have low runlengths (like 3 to 30 instructions in [85] and [77]) to exploit benefits of multithreading on a single processor system. However, L_{eff} should also be low to achieve a high U_p. In other words, n_p should be high.

In practice, a large size memory subsystem with 2 or more ports is prohibitively expensive. In comparison, on-chip caches of much lower sizes– 8 to 64 KB– on current generation processor support at most 2 to 4 ports [104]. Only for register sets, at most 5 to 10 ports are viable [61, 104]. Since the objective of multiple ports is to increase the concurrency in memory accesses, we can take advantage of the following memory organizations to serve the same purpose (listed in the order of their increasing costs):

1. A set of interleaved memory banks support concurrent accesses, if no two requests are directed towards the same memory bank. This is the cheapest approach to increase the memory bandwidth, and is widely followed. 4, 8 and 16 way interleaved memory banks are commonly observed in commercial designs, e.g. Cray X-MP/Model 24 uses 16-way interleaved memory [76].

2. A pipelined memory module can service the accesses by more than one memory requests, depending upon the number of stages in the memory pipeline. These stages can be: address decode, data read and fetch, and output data latch. This approach is costlier than using memory banks.

We briefly discuss the implications of our results on the following two design approaches:

Prime number memory system: This approach to the memory interleaving is useful to reduce contentions at memory modules, when access patterns from multiple threads are almost identical [74, 39]. A major impediment of implementing a prime-numbered memory subsystem is the increased latency for address calculation. Our results indicate with n_p concurrent accesses at the memory, the performance of the memory approaches that of the memory with a latency of $\frac{L}{n_p}$. Since the support for split-phase transaction in a multithreaded system ensures that the processor is not held up, the address computation can take place either during the context-switch time or as the

first pipelined stage in the memory subsystem. Thus, with a marginal increase in latency, access contentions can be reduced.

Cache-less Systems: Two types of general purpose computer systems, which do not use caches, have been proposed: *first*, dataflow systems [32, 47], and *second*, multithreaded systems like HEP [87, 9]. (We note that the vector computers exploit spatial locality through the use of large vector registers [76].) A common aspect between these two types of systems is that asynchronous execution along multiple threads at a processor makes it difficult to exploit the locality between the computation on these threads. So, if we attempt to exploit the locality on one thread, the miss rate on other threads may suffer [4, 91]. Cache coherence among multiple processors is another significant research problem [22]. In a cache-less system, as long as memory is capable of supporting multiple concurrent accesses, multithreading can yield substantial benefits.

These results and design considerations of a single processor system also provide an insight to the performance of a processing element in a multiprocessor system, which we discuss in the next chapter.

Chapter 5

Multiprocessor System

5.1 Introduction

In Chapter 4, we developed an analytical model to predict the performance of a single processor system. We characterized how architectural parameters like the memory access time, and workload characteristics like the thread runlength, affect the performance of the processor and memory subsystems.

A single processor system, augmented with a network interface, forms a processing node of multiprocessor systems with distributed shared memory. An analysis of these multithreaded multiprocessor systems, however, becomes significantly more complex due to an increase in the number of parameters to characterize the architectural subsystem as well as the multithreaded program workload. This chapter focuses on problems 3.2.1, 3.2.2, and 3.2.5 (discussed in Chapter 3). The objectives of this chapter are as follows. First, we extend the analytical framework to the performance modeling of such multithreaded multiprocessor systems. Second, we analyze their performance behavior. Third, we apply the analysis to optimize the processor performance on a multithreaded program workload.

A multithreaded multiprocessor system (MMS) consists of an interconnection network (IN) subsystem, and processing nodes with processor and memory subsystems. Latency for communication across the network significantly affects the performance of a multiprocessor system [14]. A multithreaded processor supports multiple outstanding accesses. The processor performs the computation on one thread, issues a long latency access, and switches

to another thread. Thus, the processor utilization improves. But a side-effect of multiple outstanding requests is to increase the contention at the memory and interconnection network, which may further increase the memory and network latencies. In turn, longer latencies for individual accesses delay the execution on the waiting threads. So, a complete performance model should capture the feedback effect of the concurrent activities at various system resources on the access rate and latency of a subsystem.

In this chapter, we extend our closed queueing network (CQN) model, developed in Chapter 4, to a multithreaded multiprocessor system. Additional considerations to the performance modeling are as follows. First, the multithreaded program workload on a multiprocessor system exhibits characteristics like the locality of accesses, i.e., whether to request a remote memory access, and if so, how far to travel on the network. Second, the number of functional units increases significantly. The interactions among functional units also needs to be modeled. Third, to accurately compute the waiting time at subsystems in the presence of access contentions, these accesses (threads) from different processor should be considered as separate customer classes in the closed queueing network. With the last two considerations, an exact solution of a CQN model is prohibitively expensive. For example, even a small system, with 2 processors having 10 threads each, has $\binom{10}{3} \times \binom{10}{3} = 1166400$ states. Here, "3" represents one processor and two memory modules on which a thread (or its memory access) receives a service.

The motivation for our performance model is to meet the needs of our anticipated users: architects, programmers and compiler writers of multithreaded systems. Given a set of architectural parameters (e.g., number of processing nodes, memory access time, network switch delay, context switch time), users would like to know:

- How to achieve high processor utilization on a target set of program workload?
- How does the processor utilization vary with the workload?
- How does the network performance vary with the program workload?

The above information should help a compiler writer for performance related optimizations on a program workload. These optimizations involve a decomposition of the data and computation on different processing nodes, and a partitioning of the computation into multiple threads. We refer to these tasks collectively as a *thread partitioning strategy*. The

performance model shows how the changes to a thread partitioning strategy would affect the performance. For system architects, the information is needed to tune the architectural parameters for a target set of workload. A concern to users of a performance model is whether they need to estimate any input parameter, which is not known *a priori*. And finally, how robust is the prediction of the performance model?

Our performance model integrates simple models of processors, memories, and interconnection network subsystems. This integrated model also accounts for the interaction between these subsystems under various application program workloads. Inputs to our integrated system model are workload parameters (e.g., number of threads, thread runlength, remote access pattern, etc.) and architectural parameters (e.g., memory access time, network switch delay, etc.). These parameters are directly provided by users. The model predicts the processor utilization, the network latency and the message rate to the network. We present the formulation of this model, its solution techniques, its verification, and its application to the performance evaluation of multithreaded multiprocessor systems.

Previous analytical studies of multithreaded architectures have mainly focused on the effect of workload parameters on processor performance (see Saavedra-Barrera et al. [80], Alkalaj et al. [8], and Agarwal [4]), but their models do not incorporate the effect of contentions at the memory and network. On the other hand, studies on interconnection networks model the effect of contentions, but assume that the effect of workload is reflected by an average value of the input message rate (see Abraham [1] and Dally [30]). We call these performance models as *open system models*, or *open queueing network* models (OQN). Perhaps the work most related to this chapter are by Willick and Eager [101], Johnson [50], and Adve and Vernon [2]. All three work model a multithreaded multiprocessor system as a closed system to capture the subsystem interactions, which we will elaborate in Section 5.11.

We construct our integrated system model based on a set of assumptions, which are reasonable for current multithreaded system implementations such as EARTH [46], TERA [9] and Alewife [6]. Our solution technique uses the mean value analysis (MVA) [75], for the following reasons. First, large systems can be efficiently analyzed. Second, the MVA is amenable to heuristic extensions for complex subsystem interactions.[1] Third, MVA provides

[1]We will discuss how such heuristics are helpful to analyze the EARTH system in Chapter 7. The *simultaneous resource possession* property, during an access at a processing node, violates the assumptions for an application of a product form solution based techniques for queueing networks.

an easy interpretation of performance measures, unlike the probabilistic approach of state-space based techniques. Fourth, it has been successfully applied to analyze real computer systems, e.g., Wisconsin Multicube [58]. We verify the correctness of the model predictions of processor utilizations, latencies and message rate, using a simulation of a Stochastic Timed Petri Net model of the multithreaded multiprocessor system. We quantitatively characterize variations in these performance measures, identify the system bottlenecks, and provide an insight to the impact of performance related optimizations.

Our closed queueing network based model enabled us to identify the *feedback* effect of subsystems on the processor performance. We show how the processor performance is affected by the maximum message rate per processor delivered by the network for a given remote access pattern (defined later in Section 5.6 as the *capacity* of the network). Our model helps a user to locate the operating point for a specific workload, and indicate whether the network capacity has been reached, thus providing a guidance on how to tune the workload parameters.

The usefulness of closed system model, like ours, is two-fold: *One*, users can work directly with the program workload and architectural parameters which are familiar to them. And *two*, when the network is near saturation, the sensitivity of performance prediction to model input parameters is significantly lower than the sensitivity of performance prediction using open system models. Our analysis indicates that the network latency reduces with a simultaneous increase in the message rate to the network because of either an increased locality in the remote access pattern or the use of a faster network! This is not obvious from open system based network studies [30, 3]. We then compare the robustness of our model with three, successively refined open system models employing feedback to improve accuracy of their processor performance predictions, and demonstrate their weaknesses and tradeoffs with respect to our model.

Finally, we show through a simple example, how a compiler writer may progressively tune a program workload. We show that the processor utilization increases with an increase in the number of threads, as long as the message rate to the network is below the network capacity, even when the observed network latency for individual accesses experienced per thread is large. Once the capacity is reached, a higher locality in remote access pattern (through data set partitioning) can improve the capacity value, and thereby improving the processor utilization. If the processor utilization remains low, then increasing the thread

runlength improves the processor utilization.

The rest of the chapter is organized as follows. In the next section, we outline our multithreaded multiprocessor system, present the multithreaded execution model, and discuss the assumptions made to develop the analytical model of the multithreaded multiprocessor system based on closed queueing network. In Sections 5.3 to 5.9 we show how to evaluate the performance of our multithreaded multiprocessor system using the results derived from the closed queueing network model. In Section 5.4 we verify the analytical predictions using Stochastic Timed Petri Nets (STPN) simulations. In Section 5.11 we discuss the related work. Finally in Section 5.12, we present the concluding remarks.

5.2 Analytical Model

In this section, we outline the multithreaded multiprocessor system (MMS), and the program workload. Subsequently, we describe how to analytically model the MMS and derive the performance measures of interest.

Our MMS consists of processing elements (PE) connected through a 2-dimensional torus, as shown in Figure 5.1. Each PE has a multithreaded processor, and a part of the distributed shared memory. A PE is interfaced to a switch on the IN. Each PE contains three subsystems: a *processor*, a *memory module* and a *switch* on the IN (see Figure 5.2). A connection exists between each pair of these subsystems at a node. An access to a subsystem incurs a delay, and may encounter a contention from other accesses.

The program execution model is similar to our earlier description in Chapter 2 (Section 2.2). The application program is a set of partially ordered threads. A thread consists of a sequence of instructions followed by a memory access. A thread repeatedly goes through the following sequence of states– *execution* at the processor, *suspension* after issuing a memory access and after arrival of response, *ready* for execution. Threads interact through accesses to memory locations. We assume that the application program exhibits similar behavior at each PE, and that the load is evenly distributed (like a Single-Program-Multiple-Data, SPMD, model) [43]. This assumption provides a user with a tangible, small set of workload characteristics to analyze their effect on performance.

The Multithreaded Multiprocessor System:

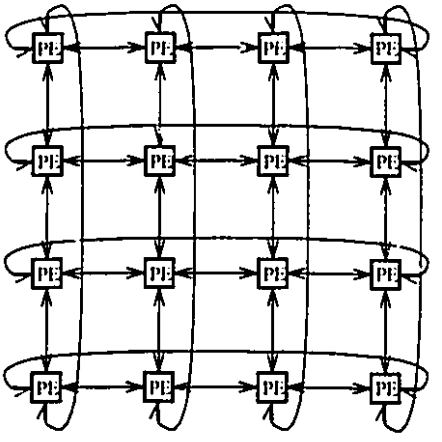


Figure 5.1: 4 x 4 Multiprocessor with 2-dimensional mesh.

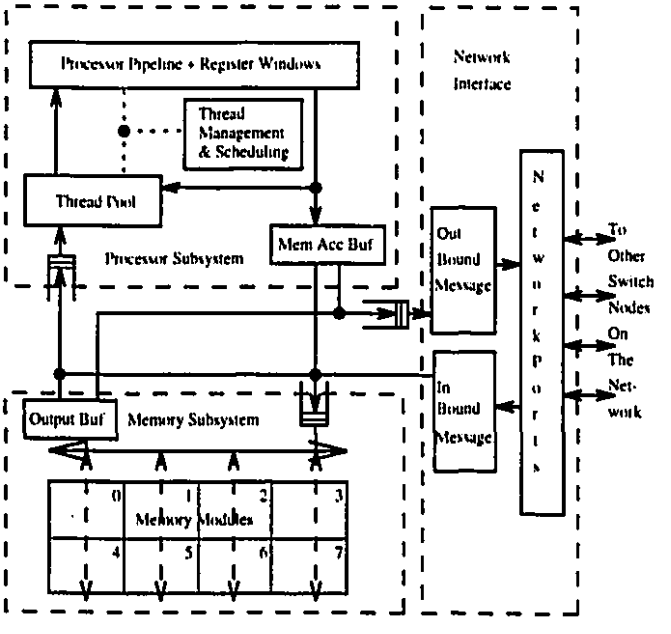


Figure 5.2: A Processing Element.

We now describe the MMS and a PE in Figures 5.1 and 5.2. Table B.1 in Appendix B summarizes all symbols. We use simple models of subsystems and show their impact on the system performance. In later chapters, we will show how subsystem interactions in a real system can be modeled and analyzed.

Processor: Each processor executes a set of n_t threads. These threads may be n_t iterations of a `do-all` loop [43]. The time to execute the computation in a thread (including the issue of a memory access) is the *runlength* of a thread, and its average is denoted by R. After issuing a memory access (whether local or remote), the processor context switches to another ready thread. C is the *context switch time*. λ_{net} is the rate of messages sent by the processor to the IN.

Memory: The processor issues a shared memory access to a remote memory module with probability p_{remote} and to its local memory module with probability $(1 - p_{remote})$. Let the memory latency, L, be the time to access a local memory module (without queueing delay) and *observed memory latency*, L_{obs}, be the latency with queueing delay at the memory.

IN Switch: The IN is a 2-dimensional torus with k PEs along each dimension. Figure 5.1 shows an example of an IN with 16 PEs. We assume that each processor is interfaced to the IN through an *inbound* switch and an *outbound* switch as shown in Figure 5.2.

The *inbound* switch accepts messages from the IN and forwards them either to the local processor or to other switching nodes towards their destination PE. The *outbound* switch sends messages from the host processor to the IN. A message from a PE enters the IN only through the *outbound* switch.

The time taken by a message between its entry to the network from an *outbound* switch and its exit from the network through an *inbound* switch at the destination PE is called the *network latency*. A message requires S time units for routing at each switch on the IN. S_{obs} is the one-way network latency with queueing delays.

The Closed Queueing Network Model:

The closed queueing network (CQN) model of the MMS is shown in Figure 5.4. Nodes in the CQN model represent the components of a PE and edges represent their interactions through messages. P, M and Sw represent the processor, memory and switch nodes, respectively. The service rates for processor, memory, and switch nodes are denoted by $\frac{1}{R}$, $\frac{1}{L}$ and $\frac{1}{S}$, respectively. An arc indicates how the access is sent from the tail of the arc

to its head. A value on an arc is the probability with which an access at the tail uses that link. Values for network links are not shown. We assume that all nodes in the performance model are single servers, with First Come First Served (FCFS) discipline. Their service times are exponentially distributed. Now, we describe how to compute waiting time at the nodes, when the system operates at steady state with n_t threads on each processor.

Processor Node: The mean value of service time for a thread on the processor is R time units. Threads do not migrate, so threads at a processor i belong to a class i in the CQN model. The waiting time $w^*_{i,i}$ for a thread at the processor node i includes its service time and the queueing delay (when other *ready* threads are serviced):

$$w^*_{i,i} = (1 + \tfrac{n_t-1}{n_t} n_{i,i})\ R \tag{5.1}$$

Here, "1" denotes the newly arrived (n_t-th) thread. $n_{i,i}$ is the number of *ready* threads at the processor i, when the number of threads in class i is n_t. Thus, $\frac{n_t-1}{n_t} n_{i,i}$ is the number of threads of class i at the processor i, when the population of class i is $(n_t - 1)$, and presents a queueing delay of $\frac{n_t-1}{n_t} n_{i,i}\ R$ to the newly arrived thread. Note that $n_{i,j}$ for class i at processor j is zero, because no class i thread is executed by processor j.

Memory Node: The mean value of service time, for each access to the memory, is L time units. For requests from a thread at processor i to the memory at node j, $em_{i,j}$ is the *visit ratio*. The *visit ratio* for a subsystem like the memory at a node j for a thread on processing node i, is the number of times the thread requests an access to memory at node j between two consecutive executions on processor i. The value of $em_{i,j}$ depends on the distribution of remote memory accesses across the memory modules. Since the service demand for a class i access at memory j is $em_{i,j}\ L$, each access waits for the following duration $w^*_{i,j}$:

$$w^*_{i,j} = (1 + \tfrac{n_t-1}{n_t} n_{i,j} + \textstyle\sum_{r=1,r\neq i}^{P} n_{r,j})\ em_{i,j}\ L \tag{5.2}$$

The term in circular brackets is the total queue length experienced by a newly arrived access from class i at the memory node j. The factor $\frac{n_t-1}{n_t}$ interpolates the queue length of class i, to reflect the queue length when the thread population in class i is $(n_t - 1)$. The queue length due to remaining classes is $\sum_{r=1,r\neq i}^{P} n_{r,j}$.

Let the distance, h, between two PEs be the minimum number of hops required along any possible path from one PE to reach another PE. Let d_{max} be the maximum distance between any pair of PEs in an MMS. We assume a geometric distribution for remote access

pattern, and characterize it using a locality parameter, p_{sw}. The p_{sw} is a factor by which the probability of an access to a remote memory module at a distance of h hops *reduces* with respect to that at a distance of $(h-1)$ hops. In particular, the probability of a memory access to a remote memory module at a distance of h hops from the local processor, is p_{sw}^h/a, where a is a normalizing constant. A low value of p_{sw} shows a higher locality in memory accesses.

$$em_{i,j} = \frac{p_{sw}^h}{a} \tag{5.3}$$

$$a = \sum_{h=1}^{d_{max}} p_{sw}^h \tag{5.4}$$

The average distance d_{avg} traveled by a remote access is:

$$d_{avg} = \sum_{h=1}^{d_{max}} \frac{p_{sw}^h}{a} \times h$$

We restrict our attention to the geometric distribution, to bring out meaningful results for users of multithreaded systems. The geometric distribution captures the locality in remote accesses, and is useful for high performance. A similar access pattern has been studied by Agarwal [3, 50]. While the remote access pattern is dependent on the workload, a locality is exploited in practice [97]. Our model is applicable to other distributions, e.g., a *uniform* distribution over P nodes, explored in Chapter 6, yields $em_{i,j}$ as $1/(P-1)$.

IN Switch Node: We model the switch as two separate nodes, *inbound* and *outbound*. The mean service time at each node is S time units. The effect of message length on service time is included in S. We make following two assumptions to simplify the switch model. The assumptions for switches are as follows: *First*, the switch operates in one direction at a time while other links are idle, e.g. Intel's iPSC/2 and iPSC/860 [12]. *Second*, the network switches are not pipelined. By changing the service rate of the switches, we obtain the desired switch performance. This method works well, except to achieve the low latency of pipelined networks in the presence of a light network traffic. This exception is not restrictive, because with as small as 4 threads, the network in an MMS is close to saturation (Section 5.6); and near the network saturation, the performance of pipelined networks is similar to that of non-pipelined networks (Appendix C).

A switch node interfaces its local PE with four neighboring switch nodes. (The mesh network has four neighbors.) The visit ratio $ei_{i,j}$, from processor i to inbound switch j, is

computed as the sum of the remote accesses, which pass through the switch j, as shown in Figure 5.3. The visit ratio $co_{i,j}$ for the outbound switch is same as $cm_{i,j}$, because all remote memory accesses of class i which pass through the outbound switch at node j are serviced by the memory at node j.

```
41: sort_distance(sorted_pes);
42:       /* sort the processing nodes according to their distance from the node (0,0),
43:       and place their "id"s in the 2-dimensional array "sorted_pes". */

44: for (d = d_max; d > 0; d - -)
45: {
46:    for (k=1; ((k<=num_procs) && (sorted_pes[d][k] = 0)); k++)
47:    {
48:       new_id = sorted_pes[d][k]; /* new_id ≡ (x,y) in the mesh */
49:       { for each neighbour of new_id {(x-1,y), (x+1,y), (x,y-1) and (x,y+1)}
50:         /* if (neighbour is more distant than new_id from processor i)
51:            {add visit-ratio of the switch at neighbour to ei_{i,new_id}} */
52:       }
53:    }
54: }
```

Figure 5.3: Pseudo code to compute the visit ratio $ei_{i,j}$.

Using the visit ratios and service times, the waiting times cor class i at switch node j are:

$$w^*_{i,j,O} = (1 + \frac{n_i - 1}{n_i} n_{i,j,O} + \sum_{r=1, r \neq i}^{P} n_{r,j,O})\ co_{i,j}\ S \quad \text{at an outbound switch} \quad (5.5)$$

$$w^*_{i,j,I} = (1 + \frac{n_i - 1}{n_i} n_{i,j,I} + \sum_{r=1, r \neq i}^{P} n_{r,j,I})\ ei_{i,j}\ S \quad \text{at an inbound switch} \quad (5.6)$$

Terms in circular brackets are the queue lengths experienced by a newly arrived access from class i at the switch node j. The factor $\frac{n_i - 1}{n_i}$ interpolates the queue length of class i, to reflect the queue length when the thread population in class i is $(n_i - 1)$. The queue lengths due to remaining classes at outbound and inbound switches are given by $\sum_{r=1, r \neq i}^{P} n_{r,j,O}$ and $\sum_{r=1, r \neq i}^{P} n_{r,j,I}$, respectively.

For a *geometric* distribution, Table 5.1 shows the visit ratios $em_{1,j}$, $co_{1,j}$ and $ci_{1,j}$ for class 1 threads at nodes 1 through 16, when p_{remote} and p_{sw} are 0.5 and 0.5, respectively. These values are computed using Equation 5.3 and Figure 5.3. (0,0) are the co-ordinates of node 1 in the 4 × 4 mesh, and (x,y) corresponds to a node j (= $4y + x + 1$). Values in **bold**-faced, *emphasis*, and brackets "□" are the $em_{i,j}$'s for the memories at distances of 1, 2 and 3 hops, respectively. Similarly, we obtain the visit ratios for other classes. With these visit ratios, we compute the waiting times using Equations 5.1, 5.2, 5.5 and 5.6.

	x=0			x=1			x=2			x=3		
y	$em_{1,j}$	$co_{1,j}$	$ci_{1,j}$	$em_{1,j}$	$co_{1,j}$	$ci_{1,j}$	$em_{1,j}$	$co_{1,j}$	$ci_{1,j}$	$em_{1,j}$	$co_{1,j}$	$ci_{1,j}$
0	0.5	0.5	0.5	**0.067**	0.067	0.183	*0.022*	0.022	0.056	**0.067**	0.067	0.183
1	**0.067**	0.067	0.183	*0.022*	0.022	0.056	[0.017]	0.017	0.033	*0.022*	0.022	0.056
2	*0.022*	0.022	0.056	[0.017]	0.017	0.033	0.033	0.033	0.033	[0.017]	0.017	0.033
3	**0.067**	0.067	0.183	*0.022*	0.022	0.056	[0.017]	0.017	0.033	*0.022*	0.022	0.056

Table 5.1: $em_{1,j}$, $co_{1,j}$ and $ci_{1,j}$ for $p_{sw} = 0.5$ and $p_{remote} = 0.5$ on a 4 × 4 mesh.

The following assumptions simplify the analysis of CQN model: (1) Every thread in a processor that is waiting for a memory request will get its request in finite time. So, there is no deadlocking of threads waiting for memory requests due to finite resources. (2) Only finite number of threads are active in a processor at any instant. This helps to avoid deadlocking at network switches due to finite buffer capacities of real network switches. (3) A sound multithreaded program execution model, such as dataflow-like EARTH model, does not have inherent deadlocks. These assumptions are reasonable for practical multithreaded systems like TERA [9], Alewife [6], and EARTH [46].

The above CQN model satisfies following conditions of a *product-form* network [15, 75]:

- *Job Flow Balance*: Since the CQN is a closed model, for each class, the number of arrivals to a subsystem equals the number of departures from the subsystem.
- *One-Step Behavior*: A change in the state of a CQN results only when there is a single access that moves between pairs of subsystems in the system. Thus, no access gets service at two or more subsystems simultaneously.
- *Device Homogeneity*: The service rate of a subsystem for a particular class does not

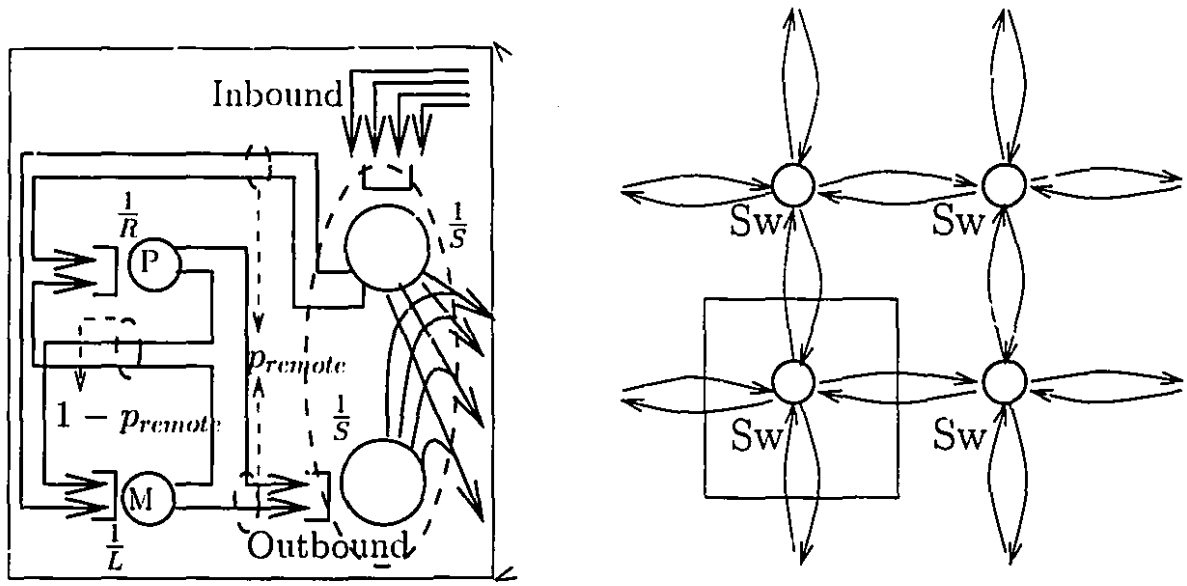


Figure 5.4: Queueing Network Model.

depend on the state of the system in any way except for the total queue length at the subsystem and the designated class's queue length.

The equilibrium state probability for a product-form queueing network can be obtained by multiplication of a function of queue lengths at each of its service centers. An explicit enumeration of complete state-space is not needed to obtain this probability. Next, we describe how to obtain the performance measures for our CQN model.

Solution Technique:

An accurate solution to the above mentioned CQN model using the state space techniques is computationally intensive [75]. For example, a two processor system with 10 threads on each processor has $\binom{10}{5} \times \binom{10}{5} = 63504$ states, where $\binom{n}{k} = \frac{n!}{(n-k)!k!}$. A solution for *product-form* networks can be obtained with efficient techniques like *Approximate Mean Value Analysis (AMVA)* [56], which is outlined in Figure A.1 (in Appendix A). The pseudo-code of the software we developed for this purpose is described in Appendix F. With n_i threads on each processor in the system, for each class i the AMVA computes:

1. λ_i, the rate at which the processor i sends memory accesses;

2. $w^*_{i,m}$, the waiting time of an access at each node m (Equations 5.1, 5.2, 5.5 and 5.6);

3. $n^*_{i,m}$, the queue length for a class i access at each node m.

The core approximate MVA algorithm computes statistics for population vectors $\mathbf{N} = (n_1, ..., n_l)$ and $\mathbf{N} - \mathbf{1_i}$. Population vector $\mathbf{N} - \mathbf{1_i}$ indicates that there are $n_i - 1$ threads at processor i, and other processors have n_l threads each. The intuition of the MVA is that a newly added thread to a class (i.e. new population $\mathbf{N}$) *sees* the queueing network in equilibrium with respect to the population $\mathbf{N} - \mathbf{1_i}$.

Step 1 of the AMVA makes an initial guess for queuelengths at each node, for P classes of threads, and a thread population of $\mathbf{N}$. Using the queue lengths at each service node for population $\mathbf{N} - \mathbf{1_i}$, waiting times are computed for the new thread/access (**Step 2**). The waiting times at various nodes are given by Equations 5.1, 5.2, 5.5 and 5.6.

Waiting times at all M nodes for a class i are used to compute throughputs (**Step 3**).

$$\lambda_i(\mathbf{N}) = \frac{N_i}{\sum_{m=1}^{M} w^*_{i,m}} \tag{5.7}$$

Using Little's law [75], queue lengths at a node for each class are computed (**Step 4**).

$$n^*_{i,m} = \lambda_i(\mathbf{N})\, w^*_{i,m}(\mathbf{N}) \tag{5.8}$$

Step 5 verifies whether the difference between the queuelengths from successive iterations of **Steps 2** to **4** are within the tolerance level. Thus, performance measures of interest are obtained at the population $\mathbf{N}$ (see the pseudo code in Appendix F).

Table B.1 in Appendix B lists all symbols used here. Based on λ_i, $w_{i,m}$, service times and visit ratios, we compute the following performance measures.

1. *Observed Network Latency*: An access from a processor i encounters queueing delay and service time at each switch in the network on its path. So, the average network latency S_{obs} for an access, is the sum of waiting time at a switch node (weighted by the visit ratio of a class i thread to that switch node) over all P switches in the IN:

$$S_{obs} = \sum_{j=1}^{P} (w^*_{i,j,I} \times ei_{i,j} + w^*_{i,j,O} \times eo_{i,j}) \tag{5.9}$$

$w^*_{i,j,I}$ and $w^*_{i,j,O}$ are waiting times at inbound and outbound switches, respectively.

2. *Message Rate to the Network*: λ_{net} is an average rate at which a processor sends accesses on the IN (to remote memory modules). λ_i is the rate at which the processor i sends memory accesses. A fraction p_{remote} of these messages are sent to remote memory modules. So,

$$\lambda_{net} = \lambda_i \times p_{remote} = \frac{n_t \times p_{remote}}{\sum_{j=1}^{M} w^*_{i,j}} \tag{5.10}$$

The denominator represents the *total wait time for an access* in class i.

3. *Processor Utilization*: λ_i is the rate at which responses to memory accesses arrive (and threads get enabled). Since R is the duration of a thread at the processor, the processor utilization U_p is given by

$$U_p = \lambda_i \times R \tag{5.11}$$

Using the above CQN model, we analyze the performance of processor and IN subsystems now. We outline these results in Section 5.3. In Section 5.4, we present the details of a simulation model based on Stochastic Timed Petri Nets (STPN) used to verify some of the analytical predictions.

5.3 Results

This section presents the highlight of the results and observations (to be detailed in Sections 5.5 to 5.10) when using our analytical model to address the performance issues of multithreaded architectures.

Without loss of generality, we analyze the MMS described in Section 5.2 as a case study with default values of some workload and architecture parameters as presented in Table 5.2. The workload characteristics are n_t, R, p_{remote} and d_{avg}. Chapter 4 shows that the effect of context switching is to limit the maximum U_p to $\frac{R}{R+C}$. So, we use R to incorporate the context switch value as well. Architecture parameters, L and S, are chosen to match the thread runlength R. The network has $k = 4$ nodes in each of the two dimensions. For architectural parameter values in Table 5.2, when the application executes one thread per

node ($n_t = 1$), and accesses are serviced by local memories, then U_p is 50%. Also, the unloaded value of S_{obs} is 27.33 time units.[2] Our results show how high S_{obs} values rise with respect to its unloaded value, under multithreaded program execution. Finally, the trends we have reported for an MMS with 4×4 mesh, are also observed for larger size systems. We report some results on the effect of scaling up to an MMS with a 10×10 mesh in the next chapter.

Workload Parameters				Architecture Parameters			Output Parameters	
n_t	p_{remote}	R	$p_{sw}(\Rightarrow d_{avg})$	L	S	k	$U_p(\%)$	S_{obs}
8	0.5, 0.8	10	0.5($\Rightarrow$ 1.733)	10	10	4	50	27.33

Table 5.2: Default Settings for Model Parameters.

Given an architecture and a program workload, our analytical model yields key performance measures of interest— the processor utilization U_p, the message rate to the network λ_{net}, and the observed network latency S_{obs}. The highlights of the results are as follows.

- In Section 5.5, our model provides a quantitative characterization of how U_p varies with the program workload and the architecture parameters.
- In Section 5.6, we explore the relation between λ_{net} and U_p, and identify the system bottlenecks for an application program. For a compiler, our model shows the impact of optimizations of the program workload on the performance, e.g. whether the network has reached the maximum number of messages it can deliver, defined later as the network capacity. If the network remains saturated after tuning the workload and U_p remains low, then the network is a bottleneck. For a system architect, such application is a test case, where the system design is not well balanced.
- In Section 5.7, we analyze how S_{obs} varies with the program workload and architecture parameters. We examine the performance of the network and processor subsystems.

[2]For comparison, a pipelined network will have a base value of 11.733 time units. However, in Section 5.6 and Appendix C we show that with multithreading and contentions, the advantage of pipelining on S_{obs} is quickly lost.

- In Section 5.8, we demonstrate the usefulness of closed system models that the program workload and architecture parameters are predictable by users. In particular, near network capacity, the network performance is highly sensitive to λ_{net}, the input parameter of open system models of the network [30, 3]. We also compare the robustness of our model with three successively refined open system models to estimate the overall performance, and show their weaknesses and tradeoffs with respect to closed system models.

- In Section 5.10, we characterize the utilizations of memory and network switches with respect to model parameters. We show how these subsystem utilizations vary when the processor performance is high. Such characterization provides an insight to the impact of program workload optimizations on the performance of various subsystems, and helps to identify bottlenecks in the system design.

We derive all analytical results using the queueing model in Section 5.2. The significant transitions in performance behavior at various places, are explained using simple bottleneck analyses.

The results are presented as follows. Section 5.4 verifies some of the model predictions using Stochastic Timed Petri Net (STPN) simulations. In Section 5.5, we characterize the processor utilization with model parameters. Section 5.6 reports an analysis of the network performance (λ_{net}), and shows the feedback effect of network subsystem on the processor performance. A large network latency S_{obs}, is considered as a fundamental cause of performance drop in single-threaded systems [14]. Hence, Section 5.7 is an investigation of the performance behavior of S_{obs}, and its impact on U_p. In Section 5.8, we demonstrate the robustness of estimating U_p by analyzing a closed queueing network model, and comparing with other approaches based on open system models. Through a simple example, Section 5.9 shows the usefulness of our results for performance related optimization of workload parameters. In Section 5.10, we characterize how utilizations of subsystems vary with model parameters. We show that an average utilization of the three subsystems at a node, called system utilization, tracks the onset of high processor performance.

5.4 Verification

In this section, we outline the Stochastic Timed Petri Net (STPN) model for the multithreaded multiprocessor system described in Section 5.2. We verify some of the predictions of our analytical model, using the simulations of (STPN) model for the MMS. The assumptions made for the STPN model are the same as those for the analytical model. While such a validation by no means is complete, simulation results provide an independent confirmation of analytical results. Simulations also permit us to study changes in architectural parameters. As noted in Chapter 2 (Section 2.3), an overlap of computation and communication poses difficulty in the run-time measurement of performance of multithreaded systems.

5.4.1 The STPN Model

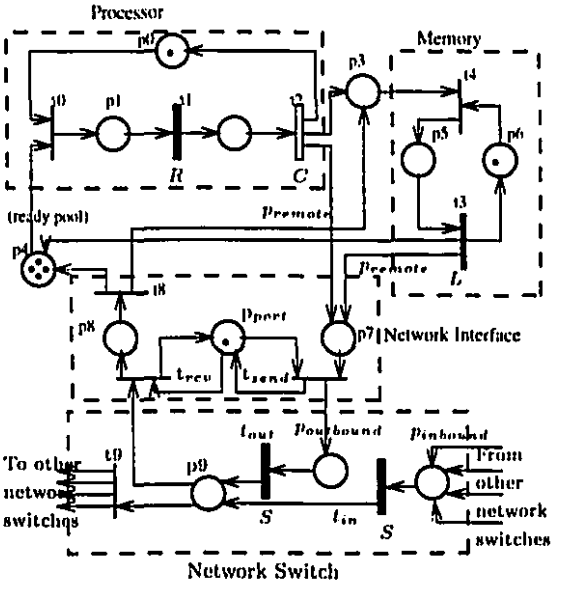


Figure 5.5: Petri Net Model for a Processing Element.

The STPN model of a multithreaded processing element (PE) is shown in Figure 5.5. A PE consists of a multithreaded processor, a memory module and a network interface. A pool of *ready* threads is maintained at place $p4$ in the processor subsystem. Transition $t1$

executes on a thread for the duration R before it encounters a long latency memory access. Transition $t2$ directs a fraction of accesses to remote memory through an *outbound* port on the network interface, with a probability p_{remote}. Otherwise, the access is serviced by the local memory. Transition $t2$ also handles saving the state of the outgoing thread and restoring that of the newly scheduled thread, in duration C.

The transitions t_{rcv}, t_{send}, and $t8$, and the places $p7$, $p8$ and p_{port}, model the network interface. An incoming message from the network for a *suspended* thread in this processor, is forwarded to $p4$, while the request to access the memory is forwarded to $p3$.

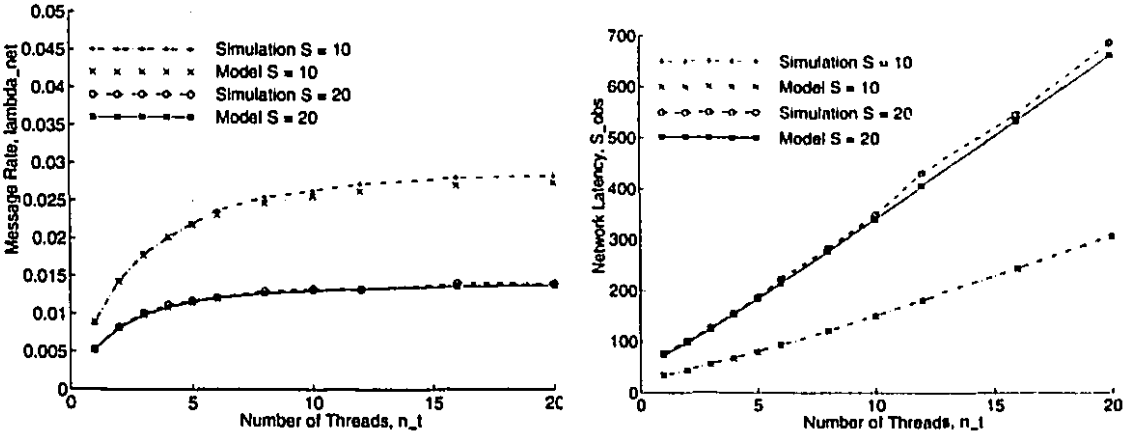
The memory port modeled by a token in $p6$, picks an access from $p3$ for service at $t3$ with a duration L. Transition $t3$ routes the response to $p4$ (local access) or $p7$ (remote access).

Transitions with *non-zero* delays are represented using rectangular boxes. R and L have exponentially distributed service time, and C has a fixed time delay.

5.4.2 Comparison with the Model

We simulated the STPN model, and also solved the analytical model for the following values of parameters: $p_{remote} = 0.5$, and $S = 10, 20$. Each run of the simulation was carried out for 100,000 time units, thus each transition fires from 50 to 5000 times depending on values of parameters. For variations with respect to n_t, Figure 5.6(a) shows the values of λ_{net}. With an increase in n_t, λ_{net} increases and reaches close to saturation by $n_t \approx 6$. For $S = 10$, the model predictions are almost identical to the simulation results. For $S = 20$, the model predictions are within 2% of the simulations. Further, model predictions in both cases are slightly lower than the simulations.

Figure 5.6(b) shows the variations in S_{obs} for the same set of parameters. We observe that S_{obs} increases linearly with n_t. While the values of S_{obs} obtained from simulations match closely with the analytical results for $S = 10$, the values are close within 5% for $S = 20$. We also studied the effect of a change in the service time distribution for memory access time (L), from exponential to deterministic, only for the STPN simulations. We found that S_{obs} values were still within 10% of original model prediction. This indicates that the small error in prediction is because of the variance in service time distributions, and demonstrates the robustness of the model.



(a) Message Arrival Rate. (b) Network Latency.

Figure 5.6: Model and Simulations.

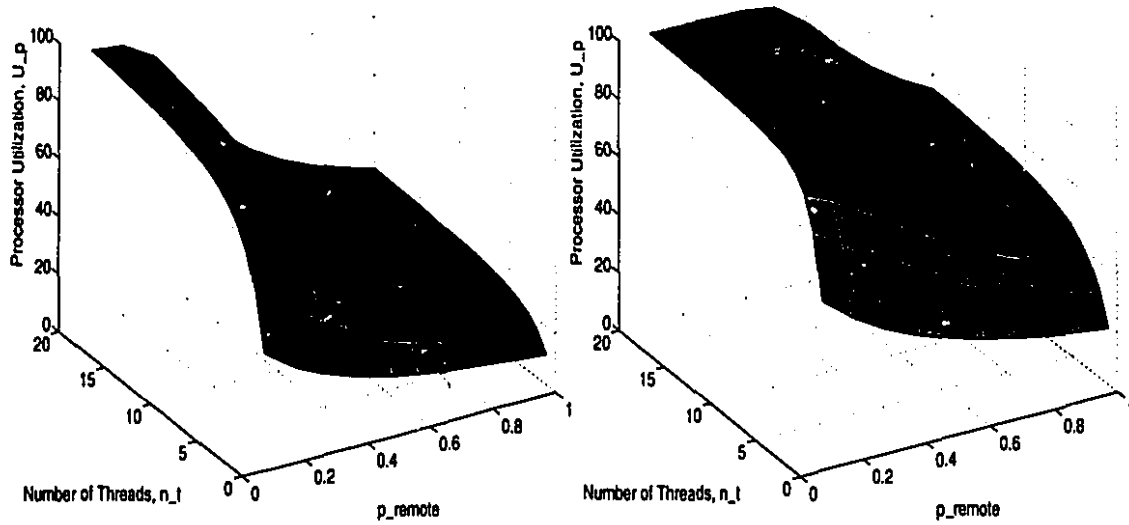
5.5 Processor Utilization

In this section, we apply our integrated system model to study the processor utilization U_p — a key performance parameter of interest. Our objective is to explore *for what values of workload parameters is U_p high?* Section 5.5.1 characterizes how U_p varies with the program workload and architecture parameters. In Section 5.5.2, we use our model to identify critical values of certain program workload parameters at which significant performance transitions occur, and verify them via an intuitive back-of-the-envelope computation based on bottleneck analysis.

5.5.1 Model Parameter Characterization

Given a set of input workload parameters and architectural parameters in Table 5.2, the AMVA yields yields the arrival rate (λ_i) of memory responses at the processor i. Since the processor is kept busy for a duration R by a thread, the processor utilization is $\lambda_i \times R$ (see Equation 5.11).

Let us consider the effect of changes in workload parameters n_t and p_{remote} on U_p shown in Figure 5.7. Expectedly, a decrease in p_{remote} and an increase in n_t yields a high U_p. Two conspicuous regions for low U_p values are, a small value of n_t, and a large value of p_{remote}. When n_t is small, the communication for a thread is not overlapped completely with the



(a) $R = 10$. (b) $R = 20$.

Figure 5.7: Effect of n_t and p_{remote} on U_p.

computation on other threads, and low U_p results. When p_{remote} is large, a larger fraction of accesses incurs network latencies. Also, increased contentions on the network increases the value of S_{obs}. (A small reduction occurs in L_{obs} at high p_{remote}.) Figure 5.7 shows that for $R = 10$ and 20, the knee of the U_p curve occurs at $p_{remote} = 0.2$ and 0.6, respectively. We analyze this transition in U_p in the next section, and investigate the network performance in Section 5.6 and Section 5.7.

The effect of network related parameters (S and p_{sw}) on U_p is shown in Figure 5.8, where n_t is 8 and p_{remote} is 0.5. Note that U_p decreases when either the switch delay S or the locality parameter p_{sw} increases. With an increase in S, a remote access suffers a large delay on each network switch. So, a *suspended* thread, expecting this response, waits for a longer duration. Note a sharp decrease in U_p when S increases beyond 10 ($= R$). With a decrease in p_{sw}, the average distance d_{avg} for a message increases, resulting in a decreased processor utilization. Since n_t is large (say 8) we expect a high U_p, but U_p still depends on the feedback from resources, which is quantified by the response time of resources. We make the following two observations from Figure 5.8: *First*, for a very fast IN (i.e. low S), the locality has no impact on U_p, because the delay for a message on the IN is negligible

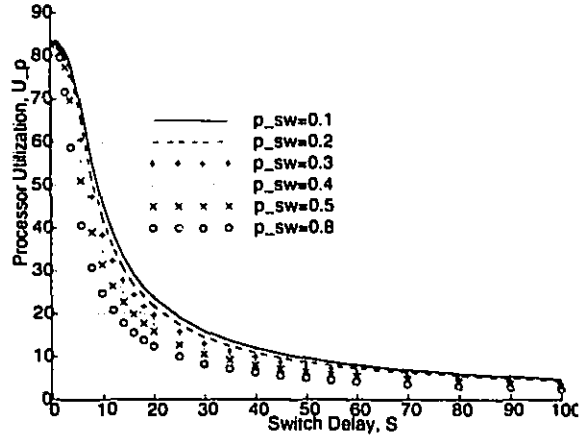


Figure 5.8: Effect of S on U_p.

compared to that on other system resources (say, memory). *Second*, a decrease in U_p due to a decrease in locality is inversely proportional to ratio of the average distance d_{avg} in two cases. For example, at $S = 10$, $\frac{U_p(at\ p_{sw}=0.1)}{U_p(at\ p_{sw}=0.8)} < \frac{d_{avg}(at\ p_{sw}=0.8)}{d_{avg}(at\ p_{sw}=0.1)} \approx 2$. Even at high S values, this observation holds, because the network is a bottleneck.

Figure 5.9 shows the effect of thread runlength R and memory access time L on U_p. To accentuate the effect on U_p, we choose a high value of p_{remote} i.e. 0.8. Let us consider two symmetric halves of this graph, say a plane at $R = L$ joining $R = L = 0$ and $R = L = 100$. On the left hand side of the (imaginary) plane $R = L$, U_p is high, elsewhere U_p is low. An increase in R has two effects:

1. A processor is busy for longer duration at higher R. So, a linear increase in U_p is observed with R in Figure 5.9. Figure 5.9 conforms with the intuition from Equation 5.11 that U_p increases linearly with R (until a high R affects λ_i).

2. The processor issues memory requests at a reduced rate (for same n_t and p_{remote} values). A reduced contention on the network helps to maintain U_p high till a larger value of p_{remote}.

Figure 5.7 also supports these two observations. A runlength of 20 yields U_p values which are twice of those for $R = 10$. Also, the knee of the U_p curve occurs at p_{remote} values of 0.2 and 0.6 for $R = 10$ and 20, respectively.

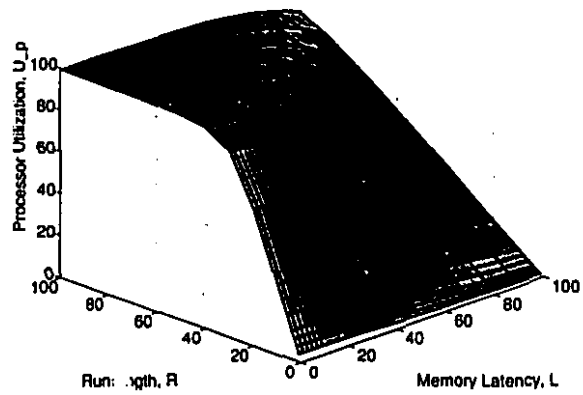


Figure 5.9: Effect of R and L on U_p.

5.5.2 Analysis of Performance Transitions

In this section we will use a back-of-the-envelope bottleneck analysis to discuss reasons for wide performance variations. We, however, note that our performance model of Section 5.2 is necessary to obtain an accurate solution.

We noted in Figure 5.7 that the knee of the U_p curve occurs at p_{remote}=0.2 and 0.6 for $R = 10$ and 20, respectively. To compute this critical p_{remote}, we apply a bottleneck analysis at the processor node in Figure 5.4. A remote access from a processor node travels $2d_{avg}$ hops for a round trip on the IN, and spends $2S$ time units to get on/off the IN. The remaining fraction of accesses, i.e. $1 - p_{remote}$, is serviced locally. Thus, message rates at the processor node are:

$$\begin{aligned} \text{message rate from processor} &\leq \text{local memory service rate} + \text{message rate from network} \\ \frac{1}{R} &\leq \frac{1-p_{remote}}{L} + \frac{1}{2(d_{avg}+1)S} \\ p_{remote} &\leq 1 + \left(\frac{L}{2(d_{avg}+1)S} - \frac{L}{R}\right) \end{aligned} \tag{5.12}$$

Equation 5.12 gives the critical p_{remote} for the knee of the U_p curve, and is independent of n_t. For a lower value of p_{remote}, the processor does not run out of threads, resulting in a high U_p. This p_{remote} value is 0.18, when p_{sw} =0.5 and $R = 10$. Figure 5.7(a) conforms with this observation. At $R = 20$, critical value rises to 0.7, which conforms with Figure 5.7(b).

When p_{remote} is increased beyond the critical value, U_p value diminishes irrespective of

n_t, mainly because either the memory or the network becomes a bottleneck.

At critical value of p_{remote}, service rates at processor, memory module, and network switch are balanced, i.e. $p_{remote} = 0.18$, when $1/R$, $1/L$ and $1/S$ are 0.1. For these service rates, a processor sends, on average, one memory access for a thread while the memory module and the network switch are busy responding to one access each. Thus, intuitively, three accesses keep subsystems busy at steady state. To tolerate differences in service times and their distributions among these subsystems, a few more threads are required. We note from Figure 5.7 that 5 to 8 threads per processor are sufficient to achieve most of the gain in U_p. Changes in service times (R, L and S) lead to changes in distribution of threads (or accesses) based on service times. This implies that performance gains are realized up to a smaller value of n_t.

5.5.3 Summary

In this section, we have shown how to compute U_p given workload and architectural parameters, characterized the processor performance with model parameters, and analyzed the significant performance transitions. The results on processor utilization U_p are as follows:

1. A high U_p can be achieved by increasing n_t as long as the fraction of remote accesses (p_{remote}) is below a critical value, which is determined by R, L, S, and remote access pattern.

2. A high R raises the critical value of p_{remote} up to which U_p remains high. Also, a processor is kept busy for longer duration when each a thread is executed, thus increasing the U_p.

With 5 to 8 threads per processor, most of the performance gains due to multithreading are achieved. When U_p remains low, to optimize workload parameters (for compilers) and architectural parameters (for system architects), an understanding of the performance of other subsystems of the MMS is essential.

5.6 Message Rate to the Network

In this section, we apply our model to study the network performance parameter of interest—λ_{net}, the message rate to the network. In Section 5.6.1, we show how to compute the message rate to the network, and its maximum (or saturation) value for a remote access pattern, the *network capacity*. We show that λ_{net} can saturate even at a low value of n_t. Hence, the performance behavior of an MMS near the network saturation is essential for performance tuning (unlike in single-threaded systems [3]). In Section 5.6.2, we quantitatively characterize the network performance behavior with workload and architecture parameters.

5.6.1 Capacity of the Network

Given the input workload and architectural parameters in Table 5.2, we solve the analytical model using AMVA. For a thread executing on a processor i, the AMVA yields the arrival rate of threads λ_i at processor i. A processor i issues one memory access for each thread, and a fraction p_{remote} are sent to remote memory modules. So, the message rate of accesses from processor i to the network is $p_{remote} \times \lambda_i$ (see Equation 5.10). Figure 5.10 shows how λ_{net} varies with n_t and p_{remote}.

Let the *maximum* number of messages delivered by the network per unit time per processor, under any remote memory access pattern, be the *maximum throughput* of the network. We also refer to this value as the *bandwidth* of the network per processor. For a particular remote access pattern, the maximum number of messages delivered by the network per unit time per processor is defined to be the *capacity* of the network. The maximum rate of messages from the processor to the network *saturates* at the network *capacity*. We denote the network capacity by $\lambda_{net,saturation}$. Since a processor allows multiple outstanding memory accesses, λ_{net} rises close to saturation even with n_t as low as 6 (see flat, dark surface in Figure 5.10 shown for $R = 10$).

We use a simple bottleneck analysis to compute the capacity. Our computation should conform with results of the AMVA algorithm, depicted in Figure 5.10. Under a remote access pattern, a message travels d_{avg} hops on the IN, and so does its response. With a delay of S time units on each switch, total duration for a message on the IN is $2\ d_{avg}\ S$ time units. At $p_{sw} = 0.5$, d_{avg} is 1.733 hops. S is 10 time units. Hence, the maximum rate at which a message is received by a processor under a remote access pattern is the capacity

of the network:

$$\lambda_{net,saturation} = \frac{1}{2\ d_{avg}\ S}\ (= 0.029 \text{ at } p_{sw} = 0.5\ ) \tag{5.13}$$

Using the performance model and Equation 5.10, we obtain λ_{net} values shown in Figure 5.10. The dark, flat surface in Figure 5.10 provides the value of $\lambda_{net,saturation}$ which is ≈ 0.029. This conforms with simple intuition given by Equation 5.13.[3]

We note that if we use only one switch instead of an *outbound* and an *inbound* switch, additional contention occurs at local switch. Thus in each direction, a message contends on d_{avg} switches (= number of hops), and one local switch. So, the network capacity is restricted to $\frac{1}{2\ (d_{avg}+1)\ S}$.

Under a remote access pattern, a message from a processing element typically travels farther than the nearest neighboring PE. The resulting value of d_{avg} would be larger than 1, and the capacity of the network is lower than the bandwidth of the IN (per processor). However, the maximum value of the capacity is the bandwidth.

To compute p_{remote} at which the IN saturates, let us consider how a PE communicates with the rest of MMS, and apply a bottleneck analysis at the *inbound* switch. A processor sends remote memory accesses through the *outbound* switch at the PE, and receives responses through the *inbound* switch. Similarly, the memory module receives remote memory accesses through the *inbound* switch, and responds these accesses through the *outbound* switch. The rate of accesses sent through the *outbound* switch at a PE equals the rate of incoming accesses through the *inbound* switch. In addition to servicing the accesses coming into the PE, the *inbound* switch also forwards accesses from neighboring switches destined for other neighboring *inbound* switches. This additional traffic is the contention on the network. This contention changes according to the locality in remote memory access pattern. Capacity of the IN is reached when the throughput at an *inbound* switch reaches $\frac{1}{S}$. Thus, a balance of message rates, at an inbound switch, yields the critical p_{remote} value for which

[3] In Appendix C, the expression in Equation 5.13 is derived using Agarwal's OQN model for a pipelined interconnection network [3]. Note that the pipelining of network links has no effect on value of $\lambda_{net,saturation}$. Also, $\lambda_{net,saturation}$ is the value at which the actual traffic in the "actual traffic vs. attempted traffic" saturates [30].

the capacity is reached:

(maximum throughput of a switch at PE i) – (forwarded messages: *contention*)

= (rate of messages from processor and memory to local switch)

$$\frac{1}{S} - \frac{\sum_{\forall r k r \neq i} ci_{r,i}}{S} = \frac{p_{remote}}{R} + \frac{p_{remote}}{L} \quad (5.14)$$

In the above equation, $\frac{1}{S}$ denotes the maximum throughput at an *inbound* switch of a PE i. Out of this total rate of messages serviced by the switch, we remove the forwarded messages (i.e. neither the source nor the destination is the PE i). Thus, the left hand side of Equation 5.14 indicates the remaining messages entering the PE i. This rate must equal the maximum rate of messages sent out by the PE, i.e. by the processor and memory subsystems. The right hand side of Equation 5.14 assumes that before the network switch becomes a bottleneck, n_t is large enough (say, ≥ 6) so that the processor and memory send remote accesses at maximum rate. At $p_{sw} = 0.5$, Equation 5.14 yields $p_{remote} = 0.3$ for λ_{net} to saturate. This p_{remote} value conforms with the prediction of the performance model shown in Figure 5.10.

5.6.2 Model Parameter Characterization

Figure 5.10 shows the effect of changes in workload parameters n_t and p_{remote} on λ_{net}, when $R = 10$. An increase in n_t as well as p_{remote} increases λ_{net}. When p_{remote} is held constant, λ_{net} increases with n_t, because more memory accesses (local and remote) are requested by the processor. A saturation of λ_{net} at high value of n_t indicates that either memory (at low p_{remote}) or network (at high p_{remote}) becomes a bottleneck. When n_t is a constant, λ_{net} increases with p_{remote}, because a larger fraction of accesses is diverted to remote memory. λ_{net} saturates, when $p_{remote} \geq 0.3$ and n_t is high (say, ≥ 6). $\lambda_{net,saturation}$ is 0.029 (flat, dark surface in Figure 5.10). Analyzing U_p values from Figure 5.7(a) with λ_{net} behavior from Figure 5.10, we note that once the capacity is reached, U_p values are low.

- An increase in n_t does not increase U_p. *What happens to additional messages at high n_t?* These messages are queued at the switches (we will see the latency values in Section 5.7).
- At a fixed value of n_t, λ_{net} remains at its saturated value even with increase in p_{remote}. But U_p decreases, because a larger fraction (p_{remote}) of accesses experiences a higher

network latency S_{obs}, in addition to the memory latency L_{obs} (at local or remote node).

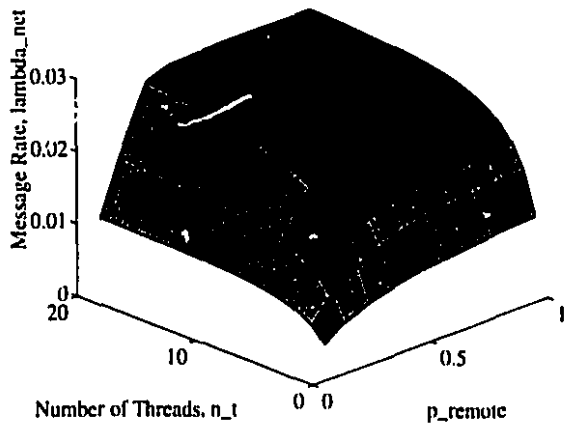


Figure 5.10: Effect of n_t and p_{remote} on λ_{net}.

The effect of network related parameters (S and p_{sw}) on λ_{net} is shown in Figure 5.11. p_{remote} is 0.5 and n_t is 8. An increase in S increases the waiting and service time for a message at each network switch. The corresponding thread at a processor is suspended for a longer duration. The longer the suspension of a thread (due to a higher S), the lower the λ_{net}. This *feedback* due to S changes the capacity of the IN. Equation 5.13 also demonstrates how λ_{net} depends on the waiting time at all nodes.

For an OQN model, the message rate is an input parameter. So, a user of an OQN model has to estimate the effect of subsystem interactions on λ_{net}. Then, the user chooses the correct network characteristics to predict the network performance. These extra steps are needed in OQN models to incorporate the effect of subsystem interactions.

We note the similarity in Figures 5.11 and 5.8. At p_{remote}= 0.8 and $n_t = 8$, the network is a bottleneck and is close to its capacity. So, U_p values track the behavior of λ_{net} with S and p_{sw}. Equation 5.10 and Equation 5.11, indicate how U_p and λ_{net} are affected by λ_i.

Figure 5.11 also prominently displays the *feedback* effect of the locality on λ_{net}. A decrease in p_{sw} implies an increase in the locality, because a message travels a shorter distance (d_{avg}) on the IN. Faster traversal of a message through the IN increases λ_{net}, and hence the capacity increases. Figure 5.11 indicates that:

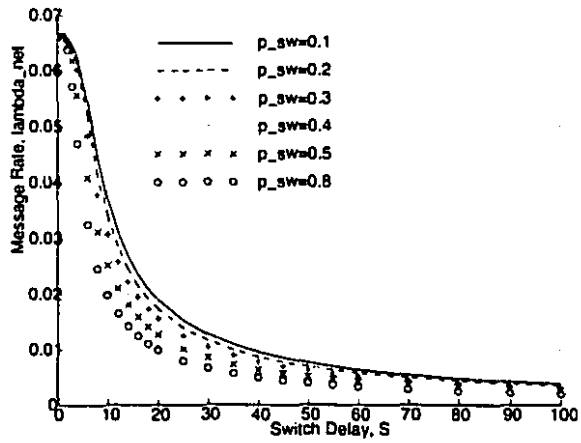


Figure 5.11: Effect of S on λ_{net}.

1. A fast network (i.e. low S) has a high capacity. On average, a fast network delivers messages too quickly, so the performance gains due to locality are negligible.

2. Equation 5.13 indicates that an improvement in λ_{net} due to an increased locality is at most of the order of the reduction in d_{avg}. Figure 5.11 shows that for $S > 6$, $\frac{maximum\ \lambda_{net}}{minimum\ \lambda_{net}} < \frac{d_{avg}\ at\ p_{sw}=0.8\ (\Rightarrow 3.17)}{d_{avg}\ at\ p_{sw}=0.1\ (\Rightarrow 1.11)}$.

Equations 5.10 and 5.11 indicate that the above observations should hold true for U_p values as well. Our observations in Section 5.5.1 conform with this intuition.

The effect of thread runlength R and memory latency L on λ_{net} is shown in Figure 5.12, where p_{remote} is 0.8 and n_t is 8. We make the following observations:

- A decrease in R reduces the time spent by a thread at the processor between successive memory accesses. So, U_p decreases, but λ_{net} increases. When the values of R and L are low (say, from $4S$ down to $2S$), λ_{net} rises rapidly to (93% of) its saturation value.

- The λ_{net} surface is symmetric with respect to R and L. For remote accesses, a thread alternates between execution at the processor, and a service at the remote memory. So, the lower of R and L values dominates the rate at which the accesses are sent on the network.

We discussed the U_p values for above parameters in Figure 5.9. Apart from λ_{net}, U_p values are also affected by R/L ratio.

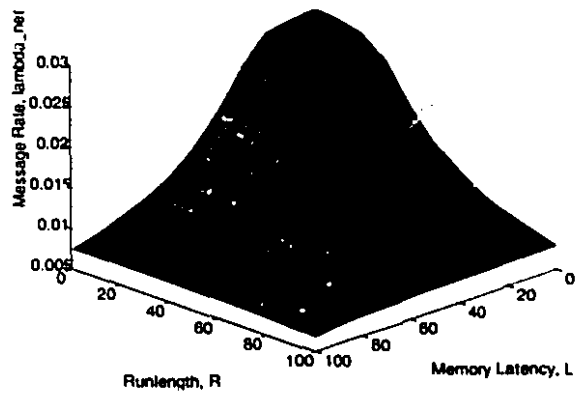


Figure 5.12: Effect of R and L on λ_{net}.

5.6.3 Summary

The main results on λ_{net}, capacity of an IN, and their impact on U_p, are summarized below:

1. When either p_{remote} or n_t is increased from a low value, λ_{net} increases, and saturates close to the capacity of the IN. At low p_{remote}, U_p is mainly governed by n_t. Once the capacity of the IN is reached, U_p does not improve with an increase in n_t, but decreases sharply with p_{remote}.

2. Capacity of the IN is determined by S and d_{avg}. A faster network, i.e. low S, has a higher capacity. (As shown in Appendix C, the pipelining does not increase the capacity.) Also, a higher locality in accesses increases the capacity. A higher capacity permits to achieve a higher U_p.

3. R and L have an identical effect on λ_{net}. An increase in λ_{net} occurs when both R and L are low. But, U_p value depends on $\frac{R}{L}$ and $\frac{R}{S}$ ratios.

5.7 Network Latency

The communication latency is considered as a fundamental cause for a decrease in performance of multiprocessor systems [14]. So, we analyze the variations of observed network

latency S_{obs} with workload and architectural parameters. We also investigate how the network latency affects processor utilization of an MMS.

5.7.1 Parameter Characterization

Given the input workload parameters and the architectural parameters, we solve the analytical model using AMVA to obtain the waiting time at each service node, for each class of threads. A remote memory access encounters service and queueing delay at each network switch in the network on its path. The *observed network latency*, S_{obs}, for an access by a thread executing on the processor i, is the average waiting time for class i threads (accesses) at each switch. An access for a class i thread does not visit all switches in the network. So, in the computation of S_{obs}, the waiting time of each switch j is weighted by the visit ratio of a class i access to that switch j (see Equation 5.9). When a model parameter changes, the waiting times at various nodes in the queueing network model changes. We perform the above computation for each set of parameter values, to reflect the change in S_{obs} value.

Figure 5.13 shows how S_{obs} varies with changes in workload parameters (n_t and p_{remote}). While S_{obs} increases with n_t, the rate of increase changes significantly with the value of p_{remote}. Using Equation 5.12 and Equation 5.13, we consider following three parts of the S_{obs} surface and the corresponding U_p values (from Figure 5.7(a)).

(i) $p_{remote} \leq 0.18$: In this region, either the processor is busy and sends accesses slowly, or the memory is a bottleneck since the most accesses are serviced locally. The messages to the IN are sent at a lower rate than $\lambda_{net,saturation}$. So irrespective of an increase in n_t, S_{obs} is close to its unloaded value. U_p values are high, when n_t is large (say ≥ 5).

(ii) $0.18 \leq p_{remote} \leq 0.3$: Beyond $p_{remote} = 0.18$, the number of messages on the IN increases. Hence, the contention as well as S_{obs} increases with n_t. Close to $p_{remote} = 0.3$, S_{obs} is high. However, the drop in U_p from its maximum value is not significant at high n_t.

(iii) $0.3 \leq p_{remote}$: Recall from Section 5.6 (Equation 5.14) that the capacity of IN is reached for $p_{remote} \geq 0.3$, thus the *inbound* switch becomes a bottleneck. When n_t is a constant, S_{obs} remains constant at a high value with respect to p_{remote}, because the number of messages on the IN (either being routed or queued) becomes a constant.

However, since at higher p_{remote} a larger fraction of messages suffer a delay of S_{obs} on the IN, U_p values decrease.

When n_t is increased, the number of accesses waiting at the network switches increase, and S_{obs} increases. The network switches are the bottleneck. So, λ_{net} and U_p do not change.

We now look at some interesting operating points of the MMS. Table 5.3 is excerpted from Figures 5.7, 5.10, and 5.13. Refer to the lines marked '*a*' and the numbers in the **bold-face** in Table 5.3. Notice that S_{obs} values 43.8, 54.8 and 82.7, for these operating points are in an increasing order with n_t, but so are their respective U_p values— 28.4%, 40.2% and 72.7%. However, when λ_{net} is close to saturation value (≈ 0.029), even a large n_t does not improve U_p. From Table 5.3, U_p values for $n_t = 4$ and 8 at $p_{remote} = 0.8$, are 26.7% and 31.5%, respectively. In summary:

1. A high U_p can be achieved on an MMS by increasing n_t, even though individual accesses may experience a large S_{obs}, *as long as* the network capacity is not reached.
2. When the network capacity is reached, even for a small value of S_{obs}, U_p remains low.

The item (1), without the mentioned condition, is an intended objective of an MMS. The item (2) suggests that when the network capacity is reached, increasing n_t has no impact. The implication of the item (2) is that we should explore other mechanisms such as increasing the thread runlength or changing the locality in access pattern.

The effect of network related parameters (S and p_{sw}) on S_{obs} is shown in Figure 5.14, when p_{remote} is 0.5, and n_t is 8. An increase in S increases the service time at each switch. So, a linear increase occurs in S_{obs}. Also through the feedback effect, U_p and λ_{net} are proportional to $\frac{1}{S}$, as shown earlier in Figure 5.8 and Figure 5.11. A similar effect is observed when the locality is changed. An increase in the locality (i.e. a low p_{sw}) decreases d_{avg} for each message, thereby decreasing the S_{obs}.

The effect of thread runlength R and memory access time L on network latency is shown in Figure 5.15. Value of p_{remote} is 0.8 and n_t is 8. Figure 5.15 shows that both parameters have a similar effect on S_{obs}. Only when both R and L are low, S_{obs} is significantly affected. For example, (even compared to λ_{net} values from Figure 5.12) S_{obs} rises sharply from 50 to

	p_r	R	n_t	L_{obs}	S_{obs}	λ_{net}	U_p	R	n_t	L_{obs}	S_{obs}	λ_{net}	U_p
	0.0	10	1	10.0	0.00	0.0000	50.00	10	2	15.0	0.00	0.0000	66.67
	0.1			10.9	29.6	0.0037	37.20			15.1	31.8	0.0056	56.30
b	*0.2*			*11.2*	*31.2*	*0.0059*	*29.36*			14.8	36.0	0.0094	46.96
	0.3			11.5	32.4	0.0072	24.13			14.2	39.4	0.0117	39.21
a	**0.5**			11.5	33.9	0.0088	17.67			**13.3**	**43.8**	**0.0142**	**28.44**
	0.8			11.3	35.1	0.0100	12.54			12.3	47.2	0.0157	19.64
	0.0	20	2	10.0	0.00	0.0000	84.53	10	4	25.0	0.00	0.0000	80.00
	0.1			13.0	30.2	0.0039	77.31			24.6	34.2	0.0074	74.07
b	*0.2*			13.1	33.2	0.0070	69.74			*22.7*	*43.5*	*0.0133*	*66.29*
	0.3			13.1	35.9	0.0093	62.30			20.0	53.4	0.0170	56.80
a	**0.5**			12.7	40.4	0.0124	49.44			**16.0**	**54.8**	**0.0177**	**40.21**
	0.8			12.1	44.5	0.0145	36.30			13.5	74.3	0.0213	26.68
	0.0	20	4	15.1	0.00	0.0000	94.68	10	8	45.0	0.00	0.0000	88.89
	0.1			15.8	31.2	0.0046	92.12			44.6	36.1	0.0086	86.26
b	*0.2*			15.9	36.1	0.0089	88.63			*40.7*	*52.7*	*0.0164*	*81.94*
a	**0.3**			15.8	42.1	0.0126	83.80			**31.5**	**82.0**	**0.0218**	**72.74**
	0.5			14.9	55.1	0.0175	70.18			19.0	120.6	0.0246	49.18
	0.8			13.3	67.5	0.0203	50.70			14.5	134.8	0.0251	31.45

Note: $p_r \equiv p_{remote}$.

Table 5.3: Performance Measures at $R = 10$ and $R = 20$.

140 time units, when both R and L decrease from 40 to 2 time units. Comparing U_p values from Figure 5.9, we note that to achieve high U_p, both $\frac{R}{S}$ and $\frac{R}{L}$ ratios should be high.

5.7.2 Summary

Above results on S_{obs} show that:

1. An increase in n_t increases S_{obs} as well as U_p. Once the capacity of IN is reached, S_{obs} increases linearly with n_t, and U_p remains constant. When the network saturates, for an n_t, S_{obs} saturates at a high value with increasing p_{remote}, while U_p decreases.

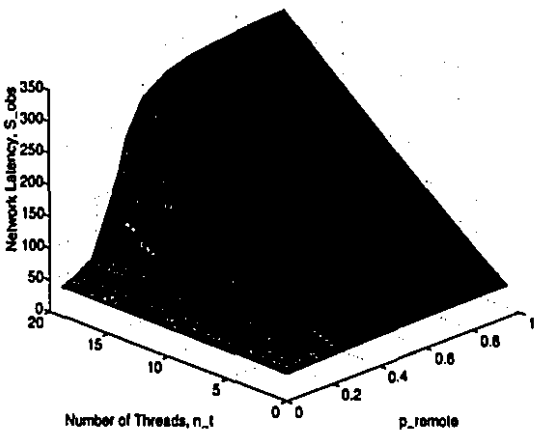


Figure 5.13: Effect of n_t and p_{remote} on S_{obs}.

2. S_{obs} increases linearly with S, and with a decrease in locality. However, a larger value of S_{obs} delays the triggering of *suspended* threads, so U_p (and λ_{net}) decreases.

3. Only when both R and L are low, S_{obs} is high. Otherwise, S_{obs} is near its unloaded value. U_p, on the other hand, depends on $\frac{R}{L}$ and $\frac{R}{S}$ ratios.

An intended objective of an MMS is the first item of increasing U_p. However, when the network capacity is reached, we should explore other mechanisms such as increasing the thread runlength or changing the locality in access pattern. Like vector machines [76, 88], with multithreading, U_p is more affected by the rate (λ_{net}) at which subsystems respond than by latencies (S_{obs}) for individual accesses. Intuitively, when the network is a bottleneck, λ_{net} saturates. On each response from a remote memory, the processor computes for R time units, and sends an access. That is, U_p depends on λ_{net} (to be precise, $\frac{\lambda_{net}}{p_{remote}} R$).

5.8 Usefulness and Robustness

In Section 5.8.1, we show the necessity to use closed system models over open system models in terms of sensitivity of performance prediction to the input parameters. In Section 5.8.2, we also compare the robustness of our model with open system models. We present three scenarios where an open system model may be applied to estimate the overall performance (such as U_p), and demonstrate their weaknesses and tradeoffs with respect to closed system

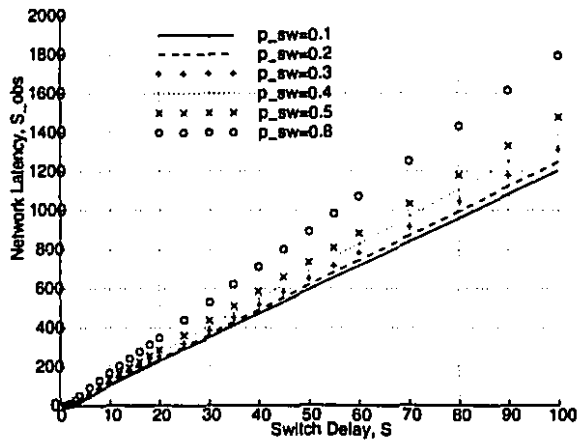


Figure 5.14: Effect of S on S_{obs}.

model.

5.8.1 Usefulness of Closed System Model

In this section, we compare the prediction of network latency using closed system model to that using open system model. We particularly focus on the region where the network is near saturation, because even with 4 to 6 threads, λ_{net} saturates (as shown in Section 5.6).

Open system characteristics for IN are reported in literature with observed network latency S_{obs} as the output parameter against message rate λ_{net} as the input parameter, e.g. Abraham [1], Agarwal [3] and Dally [30]. Initially, S_{obs} rises slowly with λ_{net}, but rises sharply when λ_{net} is close to saturation.

To draw a fair comparison of robustness in performance prediction, we also plot such characteristics. However, λ_{net}, an input parameter of the open system models, is not known *a priori*, because λ_{net} results from the interaction among various subsystems, during a program execution on a multiprocessor system. We consider two ways to vary λ_{net}: The first uses workload parameters, n_t and p_{remote}, as shown in Figure 5.16 and the second uses network related parameters, S and p_{sw}, as shown in Figure 5.17.

In Figure 5.16(b), we obtain S_{obs} versus λ_{net} characteristics, using our closed system model. The following procedure shows how to obtain the open system characteristics using our model.

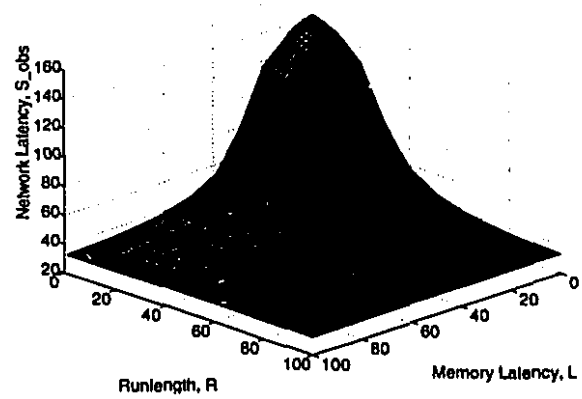


Figure 5.15: Effect of R and L on S_{obs}.

Figure 5.16(a) and Figure 5.16(c) show the effect of workload parameters on S_{obs} and λ_{net}, respectively. Figure 5.16(a) is a two-dimensional view of Figure 5.13 discussed earlier in Section 5.7. Similarly, λ_{net} behavior in Figure 5.16(c) is a two-dimensional view of Figure 5.10 in Section 5.6.

Using these Figures 5.16(a) and 5.16(c), we show a plot of S_{obs} versus λ_{net} in Figure 5.16(b) similar to the open system characteristics. To plot each point in Figure 5.16(b), we consider each pair of n_t and p_{remote} values, obtain λ_{net} value from Figure 5.16(c), and obtain S_{obs} value from Figure 5.16(a). For example, at $n_t = 10$ and $p_{remote} = 0.4$, λ_{net} and S_{obs} values are 0.0265 and 125, respectively. Each curve in Figure 5.16(a) and Figure 5.16(c) represents a fixed value of p_{remote}, and this curve gets mapped on to a small part of the curve in Figure 5.16(b), e.g. at $p_{remote} = 0.1$, λ_{net} changes between 0.0035 and 0.0095. To obtain a complete plot of S_{obs} against λ_{net}, we overlap projections for various values of p_{remote} and n_t.

Now we compare the sensitivity of the prediction of S_{obs} using OQN model in Figure 5.16(b), and using a closed system model in Figure 5.16(a). Recall from Section 5.6 that to use the MMS efficiently, the IN performance may be pushed to the network *capacity*. We assume that the operating point of the MMS is $n_t = 7$, $p_{remote} = 0.5$, $\lambda_{net} = 0.025$, and $S_{obs} = 100$. These values are close to the network saturation region. Let λ_{net}, the input parameter of the OQN model, be changed by 15% from 0.025 to 0.02875. We note that the prediction of S_{obs} changes from 100 to 290– a change of 190%! This shows that S_{obs} is

highly sensitive to λ_{net}, near the network saturation. In reality, this change in λ_{net} can be brought only if we change n_t from 7 to 20, by 186% (at $p_{remote} = 0.5$).

An estimation of queueing delay at a service node, is the reason for high sensitivity of S_{obs} to λ_{net} in the prediction using an open system model. This estimation is called as the contention factor [53, 3], because it reflects the increase in the waiting time at a service node in the presence of accesses from other processors (and threads). The contention factor is $\frac{\rho}{1-\rho}$, where ρ is the utilization of a link. Since these links are connected to each switch, we consider the utilization of a switch, ρ. Equation 5.15 shows ρ as the sum for accesses from all classes of service demand for a class at a switch times the throughput of that class. Thus, the contention factor can be expressed using λ_{net} as follows:

$$\rho = \sum_{\forall r} S \times \text{visit ratio for class } r \times \text{throughput of class } r = S \sum_{\forall r} (ei_{r,i} \lambda_{net,r}) \quad (5.15)$$

$$\frac{\rho}{1-\rho} = \frac{\lambda_{net}(S \sum_{\forall r} ei_{r,i})}{(\lambda_{net,saturation} - \lambda_{net})(S \sum_{\forall r} ei_{r,i})} = \frac{\lambda_{net}}{\lambda_{net,saturation} - \lambda_{net}} \quad \text{Contention factor} \quad (5.16)$$

In Equation 5.15, $ei_{r,i}$ is the visit ratio of class r access at the *inbound* switch i. Since S is the service time at switch i, $S \times ei_{r,i}$ is the service demand for a class r access. We substitute value of ρ in Equation 5.16. We assume that λ_{net} and $\lambda_{net,saturation}$ are same for all classes, i.e. the remote access pattern is isomorphic with respect to any processing node. Thus, Equation 5.16 can be simplified as shown in the right hand side. Near the network saturation, λ_{net} approaches $\lambda_{net,saturation}$, so S_{obs} is highly sensitive to λ_{net}.

Let us consider the same change of 15% in the input parameter n_t, of the CQN model, i.e. n_t changes from 7 to 8. The resulting values of S_{obs} and λ_{net} change by 14% to 114, and 3% to 0.0257, respectively. We note that S_{obs} varies at most linearly with n_t (Figure 5.16(a)).

The discussion above brings out the following advantages of using a CQN model for the network performance prediction, for users:

1. A user is more familiar with an input parameter like n_t or p_{remote} than λ_{net}. For example, *on how many iterations of a* `do-all` *loop should the computation begin at a time?* This is similar to the *k-bounded* computation on loops [24], and the sample program workload in Section 2.2.

2. The network performance (S_{obs}) is highly sensitive to λ_{net}, an input parameter of

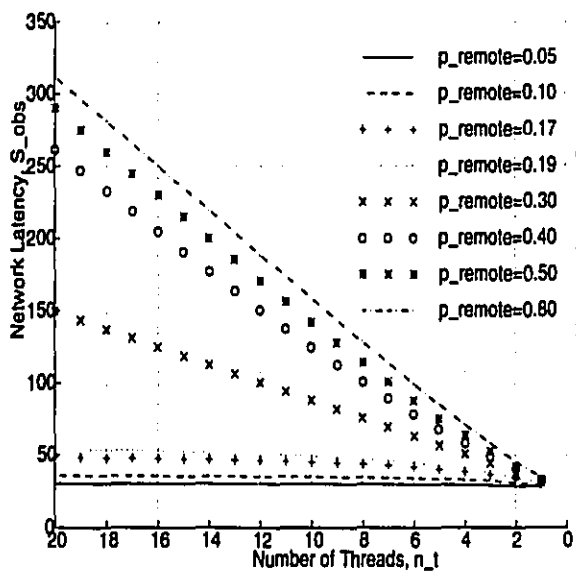
open system model, than to n_t or p_{remote}, input parameters of closed system model. This robustness of network performance prediction with respect to n_t and p_{remote} is also helpful to compute buffer requirements at network switches. For example, a small error in λ_{net} near the network saturation can lead to a significant error in buffer requirements at the switches. Further in Section 5.8.2, we also show how this robustness is helpful for processor performance prediction.

Now, we show another interesting perspective of S_{obs} and λ_{net} characteristics using network related parameters, S and p_{sw}. Figure 5.17 shows these variations, and has been obtained similar to our approach in Figure 5.16. We use $p_{remote} = 0.5$ and $n_t = 8$. Let us consider S_{obs} and λ_{net} values, when S is 10. When the locality is increased, say by decreasing the value of p_{sw}, d_{avg} decreases and S_{obs} for a message decreases, as shown in Figure 5.17(a). Since the response to a remote request is received faster, the corresponding thread is ready for execution earlier. Due to this *feedback* effect (also discussed in Section 5.6), The results is an increase in λ_{net} as shown in Figure 5.17(c). For a pair of p_{sw} and S values, we plot S_{obs} and λ_{net} together in Figure 5.17(b). In contrast to S_{obs} versus λ_{net} plot in Figure 5.16(b), we notice that λ_{net} increases while S_{obs} decreases in Figure 5.17(b). Such characteristics are not reported in OQN model based studies [1, 3, 30]. To deduce these characteristics from OQN model, a user needs to estimate λ_{net}, and then use S_{obs} versus λ_{net} characteristics with appropriate parameter settings. However, this perspective of the performance behavior due to locality variations is essential to the performance tuning techniques used by compiler writers.

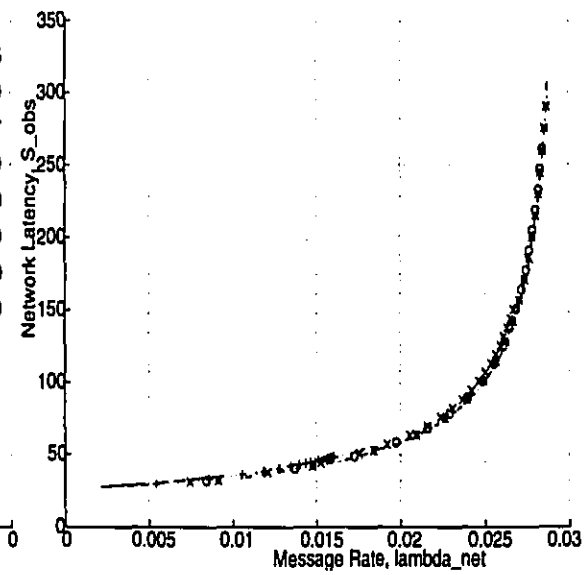
Thus, capturing the subsystem interaction helps our integrated system model to predict the effect of program workload on the IN performance (i.e. λ_{net} and S_{obs}). On the other hand, the focus of OQN studies is to evaluate the IN performance, by considering idealized interactions with the rest of the system.

5.8.2 Robustness of Processor Performance Prediction

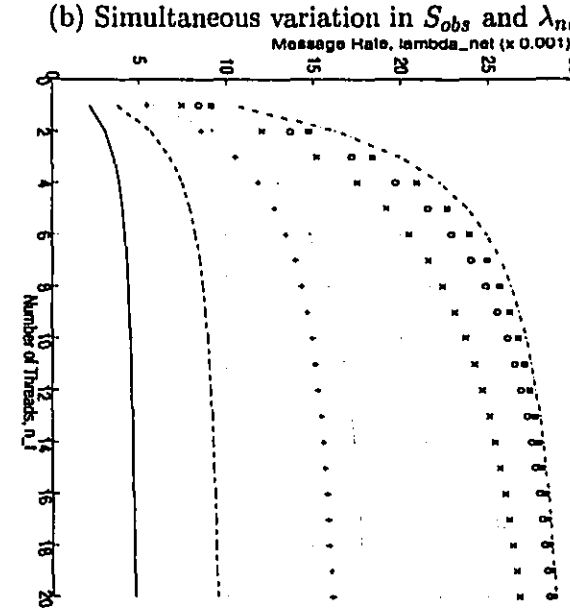
Section 5.8.1 showed that open system models are less useful than closed system models for performance prediction of multithreaded architectures, because the network capacity is reached even for small number of threads. In this section, we investigate the error in the prediction of open system models:



(a) Effect of n_t on S_{obs}.

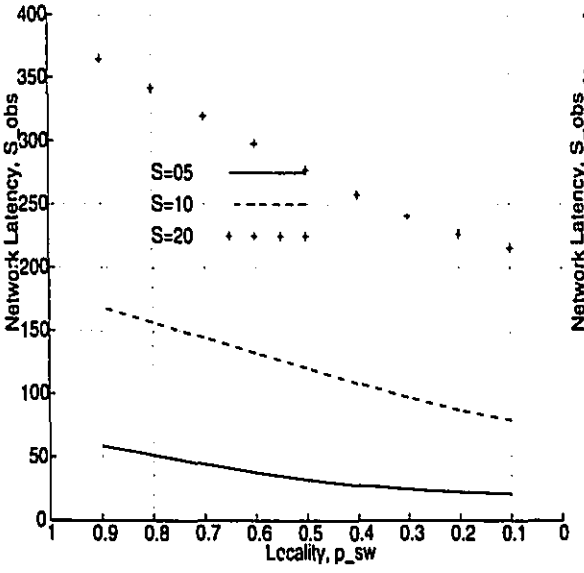


(b) Simultaneous variation in S_{obs} and λ_{net}.

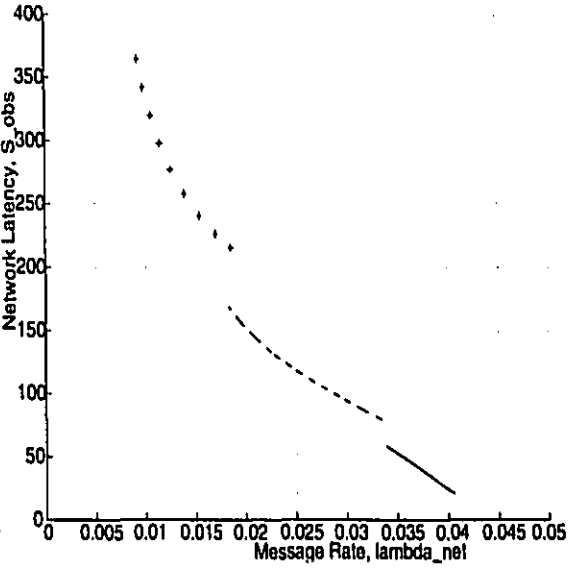


(c) Effect of n_t on λ_{net}.

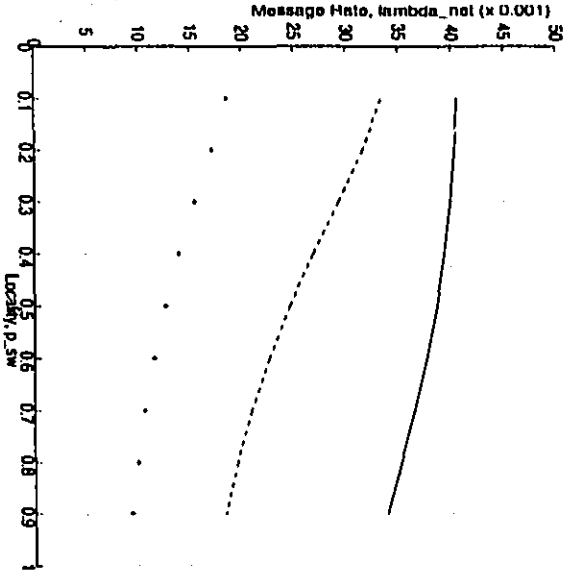
Figure 5.16: Operating Point in a Multiprocessor Network.



(a) Effect of p_{sw} on S_{obs}.



(b) Variation in S_{obs} and λ_{net}.



(c) Effect of p_{sw} on λ_{net}.

Figure 5.17: Impact of Locality on Operating Point.

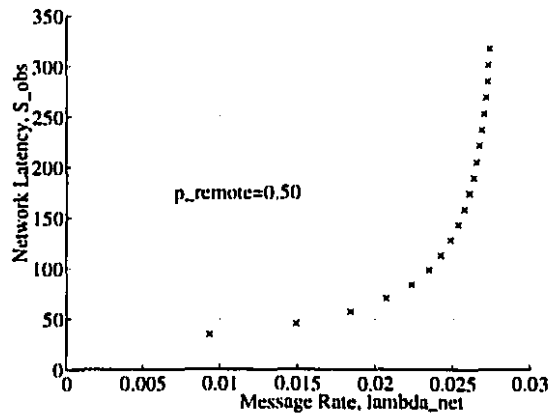


Figure 5.18: S_{obs} and λ_{net} variations at $p_{remote} = 0.5$.

What is the difference in the processor performance prediction using *closed* and *open* system models? And in case of a significant difference in the performance prediction:

Given an open system characteristics of the network, can we obtain the performance of an MMS by coupling it with a model for the processing node of an MMS?

We consider three successively refined models to predict the processor performance based on open system models, and compare their results with our closed system model. In all three cases, we assume that Figure 5.18, which is derived from Figure 5.16(b), is the representative open system characteristics of network in the MMS under study. These characteristics represent a function f, such that

$$S_{obs} = f(\lambda_{net}) \tag{5.17}$$

Next, we discuss the three models in detail.

- *The Naive Model*:

  The first model, called *naive*, uses a naive approach, which assume that the network is moderately loaded. For this load (λ_{net}), S_{obs} is obtained from the open system characteristics. Using S_{obs}, we compute the time for each thread (access) to return to the processor as ($R + L + 2\, p_{remote}\, S_{obs}$), and obtain U_p ($= \frac{n_t\, R}{(R + L + 2\, p_{remote}\, S_{obs})}$).

- *The Simple Model*:
  The second model is a *simple* model with a simplistic feedback mechanism. The model assumes initial values for λ_{net} and S_{obs}, and proceeds as the first model to compute new values for $\lambda_{net}(= \frac{n_t\ p_{remote}}{(R\ +\ L\ +\ 2\ p_{remote}\ S_{obs})})$ and U_p. New λ_{net} is used to obtain S_{obs} from Figure 5.18. The process repeats till the values in successive iterations are close enough (see Figure 5.19).

- *The Closed Loop Model*:
  The third model uses open system models of the processing node, and the network subsystem. Input and output parameters of these subsystem models are suitably interconnected to form a closed loop, and are solved iteratively. We call this model as a *closed loop model*. To capture the feedback effect, the output parameter of the processing node model is λ_{net}, which provides the input parameter of the network model. Similarly, the output parameter of the network model is S_{obs}, which provides the input parameter of the processing node model. Solving these models simultaneously yields the values of performance measures— λ_{net}, S_{obs}, and U_p.

The Naive Model:

In the *naive* model, we assume a moderate load on the network (λ_{net}), and obtain S_{obs} from the open system characteristics in Figure 5.18. We compute the cycle time for a thread (access) to return to the processor as $(R\ +\ L\ +\ 2\ p_{remote}\ S_{obs})$. This value represents the *wait time for a thread at all queueing nodes*. Each thread spends a duration R at the processor, so for n_t threads, we obtain U_p as $\frac{n_t\ R}{\textit{wait time at all queueing nodes}}$. Similar approach of an assumed, fixed network load is used by Boothe [18] and Thekkath [91], to study various aspects in multithreading.

This naive model works well when $n_t = 1$. Let us assume that S_{obs} is 27.33, i.e. its un-loaded value. Substituting values of R, L and p_{remote} in $(R\ +\ L\ +\ 2\ p_{remote}\ S_{obs})$, we obtain U_p as $\frac{10}{10+10+2\times p_{remote}\times 27.33}$ =21.1%. Our closed system model of Section 5.2 yields 17.67%.

At higher n_t values, predictions differ even widely. Let us assume that n_t is 6, and λ_{net} is 0.025 (close to the network saturation). The corresponding S_{obs} is 135. So, U_p is $\frac{6\times 10}{10+10+2\times p_{remote}\times 135} = 38.7\%$. Predictions of our closed system model for λ_{net}, S_{obs}, and U_p are 0.023, 92.5, and 46.0%, respectively. A different choice of λ_{net} (e.g., 0.02 and 0.015),

leads to widely different U_p values (69.3% and 94%, respectively). The better the prediction of network performance, i.e. λ_{net} and S_{obs}, the greater the accuracy of U_p. A weakness of the *naive* model is an absence of a feedback to improve the assumption of the network load (λ_{net}).

A caution for the use of this simple model is that the assumed load should be higher than actual, otherwise the predicted value of U_p may *wrongly* exceed 100%. For example, using an un-loaded value of S_{obs} i.e. 27.33, with $n_t = 6$ leads to a U_p value of $\frac{6\times10}{10+10+2\times p_{remote}\times 27.33} =$ 126.6%.

The Simple Model:

The *simple* model provides a feedback mechanism to improve the accuracy of the performance prediction. Starting with an assumption for S_{obs}, we use characteristics in Equation 5.17 to iteratively refine the prediction of λ_{net}, and S_{obs}. Figure 5.19 shows the steps in this model. Starting with a no-load value of S_{obs}, λ_{net} is computed as $\frac{n_t}{(R + L + 2\ p_{remote}\ S_{obs})}$, and U_p as $\frac{n_t\ R}{(R + L + 2\ p_{remote}\ S_{obs})}$. Using new λ_{net}, S_{obs} is obtained as $S_{obs} = f(\lambda_{net})$, the function depicted in Figure 5.18. The iterative process repeats till the new S_{obs} value is close to its old value. For examples we have tried, this method converged only for $n_t \leq 2$. We assumed that adjacent performance points shown in Figure 5.18 are connected using straight lines, i.e. a linear interpolation for performance points in between those shown in Figure 5.18. Predictions at $n_t = 2$, for S_{obs}, λ_{net}, and U_p, are 46.1, 0.0149, and 29.89%, while the closed system model predicts 43.8, 0.0142, and 28.44%. We note that:

- The prediction of U_p value is within 5% of that using closed system model, when network is unsaturated.
- For higher values of n_t, the iterative process did not converge. Since the steep slope of open system characteristics, the output parameters oscillated between two to three values. Figure 5.18 shows an *almost* vertical line at $\lambda_{net} \approx 0.029$ which prevents accurate computation of the network operating point (S_{obs}).
- It is difficult to study the effect of network related parameters (similar to our investigation using S and p_{sw} in Section 5.8.1) with only the knowledge of open system characteristics in Figure 5.18.

The Closed Loop Model:

Given an open system characteristics for the interconnection network, $S_{obs} = f(\lambda_{net})$

0- **Assume** S_{obs} to be a no-load value, say S_j, where j is the iteration number.
1- **Compute** λ_j and U_p, based on n_t, p_{remote}, and S_j.
2- For new λ_j, **obtain** $S_j = f(\lambda_j)$.
3- **If** $((S_j - S_{j-1}) > tolerance)$
then go to **step 1**;
else exit.

Figure 5.19: Feedback Algorithm.

The *closed loop model* uses open system models of the processing node, and the network subsystem. Input and output parameters of these subsystem models are suitably interconnected to form a closed loop, and are solved iteratively. To capture the feedback effect, the output parameter of the processing node model is λ_{net}, which provides the input parameter of the network model. Similarly, the output parameter of the network model is S_{obs}, which provides the input parameter of the processing node model. Solving these models simultaneously yields the values of performance measures— λ_{net}, S_{obs}, and U_p. The closed loop model uses analytical expression for the network performance shown by Equation 5.17. In this approach (used by Johnson [50]), a processing node is modeled using the following equation:

$$S_{obs} = \frac{n_t}{\lambda_{net}} - \frac{R}{2} \tag{5.18}$$

In Equation 5.18, the network latency is, on average, the number of outstanding remote accesses times the duration between successive responses from the network minus the execution time at the processor. A factor "2" in the last term is due to two remote messages, the request and its response, associated with each thread. Equations 5.17 and 5.18 are simultaneously solved for two unknowns, S_{obs} and λ_{net}. With the resulting value of λ_{net}, we obtain the processor utilization as follows:

$$U_p = R\lambda_{net} \tag{5.19}$$

The original model by Johnson [50] does not take into account the local memory. However, we can incorporate a detailed model of the processing node, in Equations 5.18 and 5.19.

The validity of this approach is based on the property of *aggregation* for queueing network [29]. This property states that any subnetwork (of a product-form queueing network) can be exactly aggregated into a single station with a load-dependent service rate. A load-dependent service center is the service center whose service rate is dependent on how many accesses are waiting in the queue. When remote access pattern is isomorphic with respect to the network, i.e. independent of processing nodes in networks like in 2-dimensional mesh, an analytical expression for the network performance can be obtained. Equation 5.17 represents this analytical expression. When more than one network links are needed to capture the behavior of network performance, a set of simultaneous equations may be needed to solve the model (replacing Equation 5.17).

There are two difficulties in the third model:

- When the remote access pattern (or the program workload) for all the processing nodes is not the same, an analytical expression for the network is difficult to obtain.
- When no single representative link can be found to analytically capture the performance behavior of all the links in the network, the performance model can be difficult to solve. This typically happens in an hierarchical network, for example, interconnection network of the MANNA system [38] which consists of an hierarchy of crossbar switches to achieve large configurations. In such a case, more than one representative links may be present. So, a set of simultaneous equations (one for each representative link, and one for the processing node) are needed to obtain the performance measures of the MMS. To our knowledge, such a study has not been conducted.

Another drawback specific to Johnson's model [50] is that for a program workload exhibiting a high locality (i.e. distance traveled per dimension, $k_d < 1$), the analytical expression is incorrect— a fact also observed by other researchers [79]. For this locality, the model assumes that there is *no* contention. Let us consider the parameters in Table 5.2, for which $d_{avg} = 1.733$ i.e. $k_d = 0.866$. Our results show that with $n_t = 8$, S_{obs} and U_p are 120.6 and 49.18% respectively. However, like Johnson's model, if no contention is considered on the network (i.e. $S_{obs} = 27.33$), then U_p value significantly rises to 78.85%, an error of 60%. Johnson's assumption that the network is unsaturated (and the contention is less), appears reasonable only when n_t is small (say, 1). In this case, U_p values with and without contention are 17.67% and 20.37%, respectively.

5.8.3 Summary

This section compared the performance prediction of closed and open system models:

1. A use of closed system model is robust near network saturation, and input parameters are predictable by the users. With an open system model, a small error in arrival rate near saturation can lead to significant error in determining the buffer requirements.

2. The closed system model provides a broader perspective by capturing the feedback effect, e.g., an increased locality in remote accesses decreases S_{obs}, but increases λ_{net}.

3. A comparison with three techniques based on open system models, showed that a feedback is essential for accurate processor performance prediction.

   - A simplistic feedback approach, i.e. *simple* model, for accounting the impact of the network performance on the performance of processing nodes, works well only when the network is unsaturated.
   - We point out that through a use of *aggregation* property of queueing networks, the processor performance can be computed accurately, provided the network performance can be accurately described by an analytical expression. However, such a performance model is difficult to solve in the following cases:
     (1)- when remote memory access patterns of individual, or groups of processing nodes are different;
     (2)- when the network is hierarchical, so more than one network links are needed to analytically capture the performance behavior of all the network links.
     (3)- for the specific model developed by Johnson [50], $k_d < 1$ yields an incorrect value of latency, i.e. the latency reduces compared to its no-load value.

5.9 An Example for Workload Optimization

Now, we show through a specific example (see Table 5.3 excerpted from Figure 5.7, Figure 5.10 and Figure 5.13) how a compiler writer can progressively optimize the workload characteristics to achieve high processor utilization. Our proposed order of steps is in the

order of their difficulty in implementation, however a compiler writer may prefer other orders suitable for a particular application.

As an example, consider the values marked '*b*' in Table 5.3 and are *emphasized* (Section 5.7). Let p_{remote} be 0.2 and R be 10. An increase in n_t from 1 to 4, increases U_p up to 2.3 times. λ_{net} also increases up to 2.3 times (at p_{remote}=0.2). However, a further increase of n_t to 8 yields only a small increase in U_p, when λ_{net} is close to saturation (also observe very high S_{obs} values).

At higher p_{remote} values, U_p values are low. A change of data layout (by placing the data close to computation) can improve the utilization. Table 5.3 captures this effect, when a decrease in p_{remote} sends fewer accesses to remote memory. At each value of n_t, a decrease in p_{remote} yields a higher U_p, while S_{obs} and λ_{net} decrease. However, a certain amount of data sharing is necessary, in most applications, which places a bound on the performance gains achieved by this technique.

If the processor utilization remains low, then thread runlength should be increased. We need techniques such as aggregating the network requests to the same memory module from multiple threads (works well for array accesses), increasing the number of instructions between successive memory requests by merging two or more dependent threads (at times, at the cost of parallelism), and using sophisticated register allocation/instruction scheduling techniques (which reduce the register spills). These extra efforts pay off because for higher thread runlength, the memory accesses are fewer, so the contention at the memory and the network are less (see L_{obs} and S_{obs} values). Resulting U_p values are significantly high. Note that $n_t = 4$ and $R = 20$ yields higher U_p values than $n_t = 8$ and $R = 10$. Table 5.3 also shows that for the application with $p_{remote} = 0.8$ and $R = 20$, the network is still a bottleneck for high performance. Such an application is a test case for system architects for tuning the architectural parameters.

In summary, we propose following optimizations:

1. Increase n_t to achieve a high U_p. Note that λ_{net}, S_{obs} also increase.

2. For $n_t \geq 5$, λ_{net} is close to saturation, so the performance gain due to a high n_t diminishes even though S_{obs} may not be high. Hence, increase the locality to get a higher U_p.

3. Finally, increase R to improve p_{remote} for which high U_p can be obtained. Simultaneously U_p value also increases. A high thread runlength has more impact on U_p than n_t.

Note that the order of difficulty in each of the above steps changes with the application as well as the compiler. Here we showed benefits of these steps, without discussing the costs associated with their implementations. Other alternative orderings are possible. Based on experiences in compilation of multithreaded program workloads, our performance model can be effectively applied to optimize the workload characteristics.

5.10 Subsystem Utilizations

In this section, we analyze utilizations of subsystems with changes in workload and architecture parameters. First, we discuss how the system utilization, U_{sys}, tracks the dynamics in subsystem utilizations due to variations in model parameters. Second, we use U_{sys} to point to values of input parameters, which result in high performance.

With $p_{remote} = 0$, the memory requests are restricted to the local memory module. An increase in p_{remote} increases the number of messages routed to remote memory modules across the IN. This has a two-fold effect on performance : (i) Since the latency for a remote access is higher (than the local memory latency) due to extra time spent in traversing the IN, the corresponding thread is suspended for a longer duration, (ii) A larger number of messages on the network leads to a higher contention or network congestion, which in turn increases the network latency. This reduces the utilization of the processor and memory subsystems. Figure 5.20 shows this effect of p_{remote} on the subsystem utilizations, for $L = 10$, and $L = 20$. At $L = 10$, an increase in p_{remote} from 0.2 to 0.8 reduces the values of U_p and U_m from nearly 90% to 23% and 22%, respectively. When U_{net} saturates, the fall in the values of U_p and U_m is steep. For $L = 20$, U_p and U_m decrease rapidly after the network saturates in the same way. Also, the variations in p_{remote} affect U_p and U_m identically.

Similar observations could be made, when we consider the effect of memory latency on the processor and network utilizations, or the effect of S on the processor and memory utilizations. If the memory latency is increased then the number of requests waiting at the memory increases, thereby reducing the values of U_p and U_{net}. Similarly, an increase in S

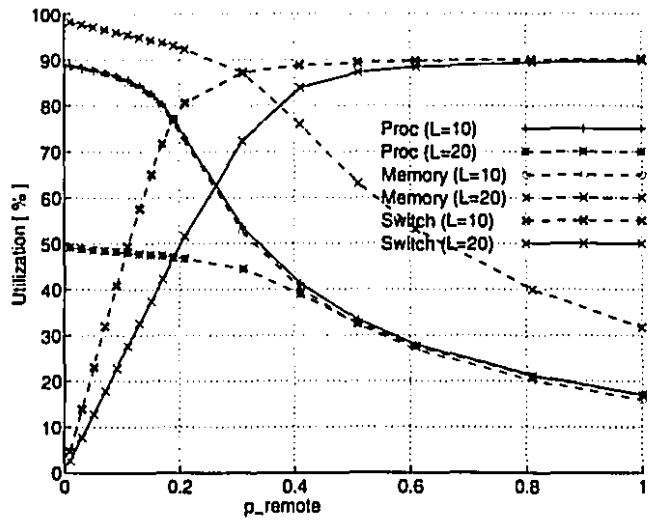


Figure 5.20: Subsystem Utilizations.

increases the network latency, so more threads at the processor remain suspended waiting for the corresponding memory responses to arrive. In turn, this decreases the rate at which memory accesses are sent, resulting in a fall in the values of U_p and U_m with respect to an increase in S.

Thus, we observe a close coupling among the subsystems, based on our integrated model of processor, memory and network subsystem.

System Utilization

Having known the behavior of subsystem utilizations (from Figure 5.20), we are interested in the ability of U_{sys} to track the transitions corresponding to saturation of these subsystems. Figure 5.21 plots the subsystem utilizations and U_{sys} with respect to memory latency for $R = 10$, and $R = 20$. When L is close to *zero*, the system utilization is low due to the low utilization of memory. At values of L close to 100, the memory subsystem saturates, but the U_{sys} is low (the limiting value is 33%) due to low U_p and U_{net}. For U_{sys}, a peak occurs when $L = R = S(= 10)$, since all subsystems are close to their maximum utilization values. With $L > 10$, both U_p and U_{net} drop off sharply with L, and only a small rise occurs in U_m,

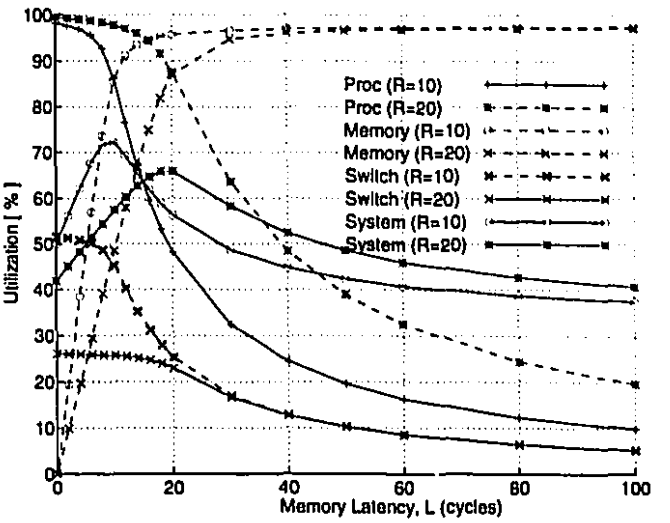


Figure 5.21: System Utilization with respect to L.

resulting in low value of U_{sys}. The maximum value of U_{sys} is referred to as the *peak system utilization (PSU)*. Let the corresponding memory latency be L_{PSU}. From Figure 5.21 we observe that:

(i) U_{sys} reflects the relative values of U_p, U_m and U_{net}. When parameters of processor and memory subsystems are considered, PSU occurs at $L = R$. We note that PSU represents a transition phase in which one subsystem approaches saturation and utilizations of other subsystems drop. This is due to balance of throughput between any pair of subsystems.

(ii) For $R = 10$ and $L \leq 10$, at PSU, U_p is only 5% less than its maximum value while U_{sys} has improved by almost 25%. For $R = 20$ and $L \leq 20$, these differences for U_p and U_{sys} are 7% and 30%. Thus, by keeping the operating range near PSU, we gain considerably in overall system utilization and the loss in processor utilization is small.

(iii) For any value of L less than L_{PSU}, U_p is high. Thus, L_{PSU} represents the slowest memory we can operate without hampering a high system performance significantly.

The bell shaped plot for system utilization also occurs with respect to changes in other parameters such as R and S.

Effect of Network Parameters

Figure 5.22 shows the effect of p_{remote} on the system utilization for various values of S. Curve for each value of S is bell-shaped, with a PSU occurring at $p_{remote,PSU}$. For $p_{remote} \leq p_{remote,PSU}$, U_p is high and the network is unloaded. U_{sys} increases with p_{remote}, because more messages get diverted to remote memory modules across IN. For $p_{remote} \geq p_{remote,PSU}$, most of the messages wait at the IN, and U_p decreases rapidly. Consequently, U_{sys} also decreases for high p_{remote}. We observe that : (i) PSU lies between 70 to 80% for a wide range of S, and (ii) For faster switches i.e. low S, U_{net} does not saturate until p_{remote} is high. Thus, a large number of messages can be transported without congesting the network.

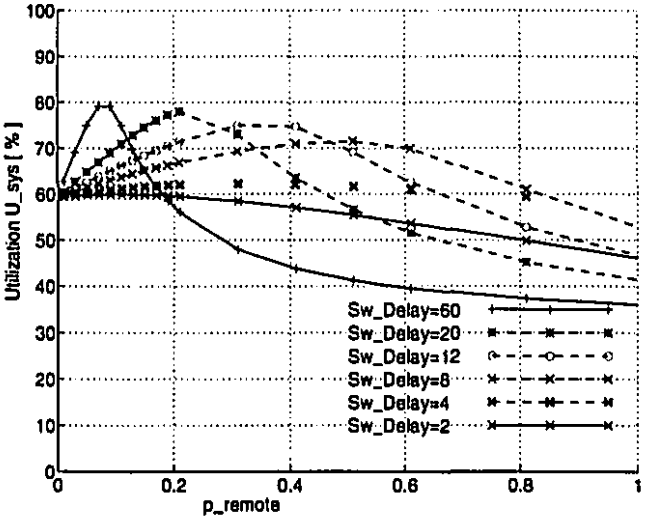


Figure 5.22: Effect of p_{remote} on U_{sys} for various S.

Effect of Thread Runlength

Figure 5.23 plots the system utilization with respect to p_{remote} for various values of thread runlength. Let us assume that the average time taken by a message on the unloaded network to complete a round trip is T_{avg}. For a geometric distribution of memory accesses with p_{sw} = 0.5, a remote memory access travels a distance d_{avg} = 1.733 hops on a 4×4 mesh. Thus,

a round trip takes 2× 1.733× 10 time units in the unloaded network. In addition, a delay of S (= 10) time units is incurred at the local switch on the forward as well as the return path of the message. Hence T_{avg} (= 34.66+ 20 = 54.66) is given by :

$$T_{avg} = 2(d_{avg} + 1)S \tag{5.20}$$

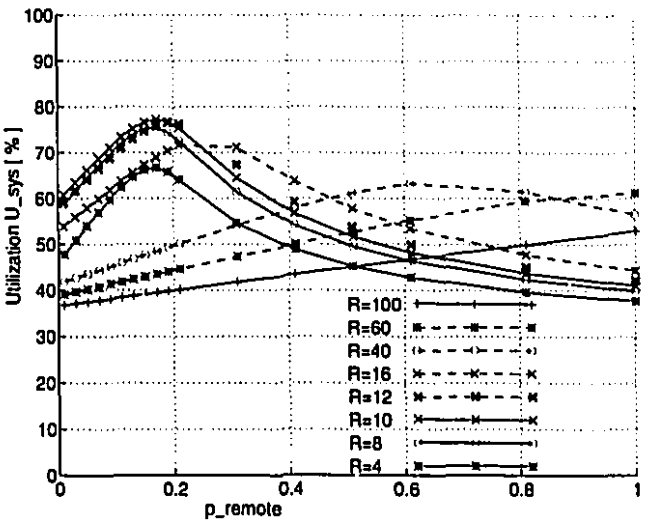


Figure 5.23: Effect of p_{remote} on U_{sys} for various R.

In Figure 5.23, for $R \leq 10$, PSU increases with R from 67% to 79%. Also, the PSU almost always occurs at $p_{remote} \approx 0.18$. Since $R \leq L$, a thread spends less time at the processor than it spends at the memory module, PSU results from the matching of throughput between the memory and network subsystems. A memory module returns the remote memory accesses to the network at the rate of $\frac{p_{remote}}{L}$. At PSU, throughput of the incoming messages from the network (= $\frac{1}{T_{avg}}$) equals the throughput of the responses from the memory module (= $\frac{p_{remote,PSU}}{L}$). So, $p_{remote,PSU}$ is $\frac{L}{T_{avg}} = 0.18$. For $R \geq 10$, the processor and network subsystems govern the PSU value. A processor sends out memory requests at the rate of $\frac{1}{R}$. A fraction (=p_{remote}) of these are directed across the network to remote memory modules. The network delivers the messages to processor at the rate of $\frac{1}{T_{avg}}$. As the throughputs should match at PSU, p_{remote} should equal $\frac{R}{T_{avg}}$. Considering these two

scenarios together, the maximum value of PSU occurs when throughputs of the three subsystems are equal. That is, the thread runlength, memory latency and network latency should be such that:

$$\frac{p_{remote,PSU}}{R} = T_{avg} \quad \text{at PSU}$$
$$\frac{p_{remote,PSU}}{L} = T_{avg} \quad \text{at PSU} \tag{5.21}$$
$$\frac{1}{R} = \frac{1-p_{remote}}{L} + \frac{1}{T_{avg}} \tag{5.22}$$

Equation 5.22 results from Equation 5.21, because at steady state memory access rate from a processor to its local memory ($= \frac{1-p_{remote}}{R}$) is matched by the service rate of the local memory. The remaining fraction p_{remote} is serviced by the network. When one subsystem saturates, Equation 5.21 could be applied to obtain the utilization values for other subsystems. For example, in Figure 5.20, we observe that on network saturation the values of U_p and U_m are close to $\frac{R}{p_{remote} \times T_{avg}}$ and $\frac{L}{p_{remote} \times T_{avg}}$. Similarly, in Figure 5.21 when the memory subsystem reaches saturation, the values of U_p and U_{net} are proportional to $\frac{R}{L}$ and $\frac{S}{L}$, respectively. We note that Equation 5.22 is same as Equation 5.12 obtained in Section 5.5.2.

Locality of Memory Accesses

If the remote memory access pattern is a geometric distribution, an increase in p_{sw} increases d_{avg} for a message on the network, and hence the network latency. Figure 5.24 shows the effect of increasing p_{sw} on system utilization, for various values of thread runlength when $p_{remote} = 0.17$. For low value of p_{sw}, PSU occurs due to saturation at processor and memory subsystems. PSU increases from 65% to 78% when p_{sw} is increased from 0.1 to 0.7 due to an increase in the value of U_{net}. Further increase in p_{sw} to 0.9 brings down PSU to 72%, due to lower values of U_p and U_m.

Summary

Our study suggests the following conditions for achieving high performance:

- Overall high utilization of all subsystems is achieved irrespective of the value of n_t (> 1), when (i) the thread runlength R equals the memory latency L; and (ii) the

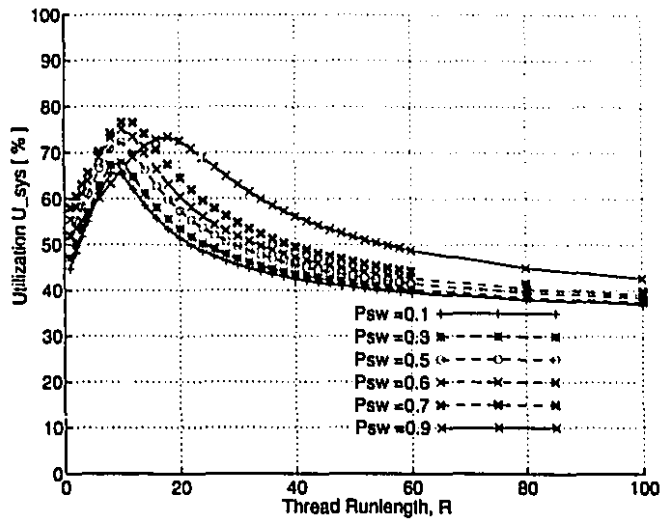


Figure 5.24: U_{sys} with Geometric Distribution.

remote memory access rate ($\frac{p_{remote}}{R}$) equals the network service rate $\frac{1}{T_{avg}}$.

- The applications with larger locality can tolerate slower networks without much degradation in performance due to reduced network traffic.

5.11 Related Work

A number of analytical and simulation studies on the performance of multithreaded architectures have been reported in the literature. First group of analytical studies focuses on the processor performance only (e.g. Saavedra-Barrera et al. [80], Alkalaj et al. [8], and Agarwal [4]), while second group studied interaction of various subsystems in an MMS (e.g. Johnson [50], Nemawarkar et al. [66], and Adve et al. [2]).

Saavedra-Barrera et al.[80] and Alkalaj et al. [8] use Petri Nets to analyze multithreaded systems. Their analysis uses the state space of PNs. Since contentions at the memory and network increase the state space tremendously, the contentions are not studied. These models are similar to the *naive* model in Section 5.8.2.

Agarwal [4] presents an analytical model for processor performance using cache parameters. The model is similar to the *closed loop* model in Section 5.8.2, except that the effect of latency on cache miss rates is not included and the memory is not modeled. So, the model is applicable, where latencies and cache miss rates are independent, e.g. either latency or cache miss rates are low.

The interconnection network performance has been extensively analyzed using open queuing network (OQN) models in the literature [3, 30]. We showed that an OQN model does not properly capture important subsystem interactions such as the feedback effect in an MMS, hence the network performance predictions of these studies were not directly usable for analyzing MMS. However, note that OQN models provide an insight to the IN performance, with a minimal set of assumptions about the system and the program execution model.

The performance model for an IN can be effectively combined with models for other subsystems to capture their interactions. Four studies using closed system models follow this approach [101, 50, 66, 2]. Our work and these studies, complement each other to reinforce the claim that a CQN faithfully models subsystem interactions of a large scale MMS.

Johnson [50] develops a closed system model to account for the feedback effect of the IN, and predict the effect of locality in an MMS. This is the *closed loop model* in Section 5.8.2. He adds the model of a program execution to Agarwal's network model [3]. Johnson's model assumes an unsaturated network and does not take into account the effects of the memory subsystem. His model does not capture the network behavior correctly, if the locality in the remote access pattern is high (specifically $k_d \leq 1$). His results show benefits of tuning the workload parameters to exploit the locality, which we have confirmed in this paper.

Willick and Eager [101] studied the performance of k-ary n-cube interconnection networks embedded in multiprocessor systems. The focus of their study is to present a performance model for such an interconnection network, with each processing node allowing multiple outstanding requests. They have not analyzed the system performance in detail. Thus, their results do not bring out specific hints for users to optimize the performance of multithreaded systems.

In a recent, independent work, Adve and Vernon [2] also use a CQN model to analyze the

performance of a k-ary n-cube network in an MMS. They focus on effects of architectural features (i.e. pipelining and virtual channels in the IN) on performance. They develop a new set of approximations for AMVA. The model increases in complexity, but provides a greater accuracy. In contrast, we use a simple network model, which reduces the complexity of the model, and use a well known approximation to apply AMVA. Our results show the effectiveness of multithreading to tolerate long latencies. In particular, we have identified the role of network capacity on the network latency and processor utilization.

Three simulation studies also report the performance benefits of multithreading [100, 18, 90]. An early study by Weber [100] shows the differences in performance gains due to multithreading, because of variations in the bus traffic. Thekkath [90] studies the effectiveness of multithreading in presence of cache. Their results indicate the need for tuning of workload characteristics such as the locality, and number of threads, to obtain performance gains using multithreading, which is validated by our results. Boothe [18] shows the benefits of various techniques manipulating the network messages, adjusting the thread runlengths, for multithreading. While confirming these results, we also show that their assumption of a constant network latency [91, 18] is not realistic, so the degree of performance gains using proposed techniques, changes substantially.

5.12 Conclusions

In this chapter, we proposed a performance model for analyzing a multithreaded multiprocessor system. Our integrated system model, based on closed queueing networks, takes into account the behavior of processors, memories and interconnection network, and the interaction between them under various program workload.

Given program workload and architectural parameters, we showed how to derive the key performance measures– processor utilization U_p, message rate to the network λ_{net}, and observed network latency S_{obs}. We applied our model to provide a quantitative characterization of their variations with model parameters, to understand system bottlenecks, and to provide insight to the impact of performance related optimizations.

Our analysis brought out the importance of *feedback* effect of network performance on processor utilization. We showed that a strong coupling exists between these subsystems. A variation in parameters of one subsystem affects utilizations of other subsystems as

well. Concurrently analyzing the network and processor performance we also showed the significance of *network capacity* to tune the workload characteristics in order to achieve a high processor utilization. For example, U_p increases with an increase in the number of threads (n_t) as long as the capacity of the IN is not reached, even when S_{obs} is large. Also, the higher the capacity, the higher can be the U_p value.

We demonstrated the usefulness of closed system performance models to users (compiler writers, programmers, and system architects): they can work directly with the program workload parameters and architectural parameters which are familiar. Added advantage is that near the network saturation, unlike open system models, problems of dealing with high sensitivity of performance to input parameters, do not arise. We also showed that the robustness of our model is helpful in processor performance prediction, in comparison to three successively refined approaches based on open system models to estimate the processor performance.

In the next chapter, we will explore how effective is the multithreading in tolerating long latencies. Then in Chapters 7 and 8, we will apply our performance model to analyze McGill's EARTH-MANNA multithreaded multiprocessor system.

Chapter 6

Latency Tolerance

The previous chapter showed how, given a multithreaded architecture and a program workload, to derive absolute performance measures, like processor utilization. We discussed how the model parameters affect the processor utilizations in a multithreaded multiprocessor system. We also noted the effect on the message rate to the network and the network latency. These performance measures, however, do not provide an estimate of the performance loss due to latencies at subsystems, e.g. memory. Users of multithreaded architectures may spend large efforts to tune the performance. For example, a system architect needs to explore a large design space to tune the subsystem implementations or the system configuration. Similarly, a compiler needs to change a significant number of workload characteristics to achieve performance improvements.

This chapter focuses on quantifying how effective is the multithreading technique in tolerating long latencies for memory accesses. We restate the following problem 3.2.3 from Chapter 3:

Problem 6.0.1 *Given a multithreaded architecture and a program workload:*

1. *Can we quantify the* latency tolerance*?*
2. *How does the ability of latency tolerance vary with model parameters?*
3. *How is the ability of latency tolerance related to the high processor performance?*

Thus, the objectives of this chapter are, to quantify the latency tolerance, and to show the usefulness of latency tolerance in performance optimizations.

The benefits of quantifying the latency tolerance are as follows. There are many workload and architectural parameters, which affect the performance of a multithreaded system. With information on tolerating particular latencies, like the network latency, a user can narrow the focus to tune the parameters, which have a large effect on the system performance.

On a target set of workloads, a system architect experiments with the system configurations, e.g. the number of processing nodes, and number of concurrent memory operations, and architecture parameters, e.g. routing delays at switches. The latency tolerance shows how changes in these parameters affect the performance, thus bringing out the performance bottlenecks. For example, if the latency of a memory subsystem is less tolerated (than say the network latency), then a system architect can tune the memory subsystem. Tuning the parameters of other subsystems will have less effect on performance.

Given a multithreaded multiprocessor system, a compiler writer has to optimize a program workload. The number of threads, their granularity, and the locality in their remote accesses, are the typical program workload parameters for optimization. A characterization of the latency tolerance with workload parameters helps to choose an effective thread partitioning, i.e. a suitable computation decomposition (thread partitioning) and data distribution. For example, if network latency is not *tolerated*, then a compiler can redistribute the data and computation to reduce the messages on the network. Changes in the number of processors in the system have a significant effect on performance of particular thread partitionings and data distributions. The latency tolerance can also be used to analyze one or more subsystems at a time.

We are not aware of any literature that quantifies the latency tolerance as a measure to evaluate system performance. Perhaps the only related work, of which we are aware, is by Kurihara et al. [54]. The authors show how the memory access costs are reduced with the use of 2 threads (per processor). Our conjecture, however, is that memory access cost is not a direct indicator of how well the latency is tolerated.

Intuitively, we say that a latency is *tolerated*, when the progress of computation is not affected by a long latency operation. In other words, if the processor utilization is not affected by the latency for an access, then the latency is tolerated. The latency tolerance is

quantified using the *tolerance index* for a latency, and indicates how close the performance of the system is to that of an *ideal* system. An *ideal* system assumes the value of the latency to be *zero* time units. Under this assumption, the performance of a system is independent of scaling. We do not assume that an *ideal* subsystem is a *contention-less* subsystem with finite delay, because the performance of such a system is likely to change, when either the number of processing nodes or the data distribution is changed.

We compute the tolerance index using the analytical framework developed in Chapter 5. Analytical results are obtained for a multithreaded multiprocessor system (MMS) with a 2-dimensional mesh. A characterization of the tolerance index with various architectural and workload parameters, helps us to tune the system performance.

We begin the next section with a discussion on latency tolerance. In Section 6.2, analyzing the latency tolerance on an MMS, we show a strong impact of the memory and IN subsystems on the system performance. Further, we apply latency tolerance to analyze a thread-partitioning strategy. This strategy deals with how to partition the computation in a `do-all` loop in terms of the number of threads and their granularities, as discussed in Chapter 2. We show that the latency tolerance does not depend on the actual latency incurred by individual messages, but on the rate at which the subsystems can respond to remote messages. In other words, a high value of a latency for messages does not imply a degradation in system performance.

Tolerating a memory (or network) latency indicates that the memory (or network) subsystem is not a bottleneck. Our analysis of memory latency tolerance in Section 6.3 shows that to ensure a high processor performance, however, it is necessary that *both* the network and memory latencies are tolerated. With a small number of threads, performance gains saturate due to a low hardware parallelism (per processor). Use of mechanisms like pipelining/multi-porting at the system resources (like memory), boosts the performance gains up to a higher number of threads. However, increasing the thread runlength for a small number of threads (> 1) results in the best performance.

The above results use an MMS with 4×4 mesh, similar to our study in Chapter 5. The default parameters are given in the Table 6.1. In Section 6.4, we analyze the tolerance index when the number of processors is scaled from 4 to 100, i.e. the number of processors in each dimension, k, varies from 2 to 10. We show that a *geometric* distribution performs

significantly better than a *uniform* distribution for larger systems, because with a *geometric* distribution, the messages cover a smaller average distance on the IN. In a large system, various switches on the network can act as pipeline stages for remote memory messages from a processor. These stages present finite delays to the messages, thereby reducing otherwise severe contentions at the remote memories. As a result, under a suitable locality, the performance of a system with finite switch delays is better performance than even an *ideal* (very fast) network.

From Section 6.1 to Section 6.5, we analyze and discuss these results. In Section 6.6, we compare our work with the related work. Finally, Section 6.7 concludes this chapter.

Workload Parameters				Architecture Parameters			
n_t	p_{remote}	R	$p_{sw}(\Rightarrow d_{avg})$	L	S	k	n_p
8	0.2, 0.4	10, 20	0.5($\Rightarrow$ 1.733)	10, 20	0, 10	4, 2–10	1

Table 6.1: Default Settings for Model Parameters.

6.1 Tolerance Index: A Metric for Performance Analysis

In this section, we discuss the intuition for latency tolerance and define the *tolerance index* to quantify the latency tolerance.

When a processor requests a memory access, the access may be directed to its local memory or a remote memory. If the processor utilization is not affected by the latency at a subsystem, then the latency is *tolerated*. That is, the latency at a subsystem does not lead to any additional idle time at the processor. Two possible reasons are, either the subsystem does not pose any latency to an access, or the processor progresses on additional work during this access. In general, however, the latency to access a subsystem delays the computation, and the processor utilization may drop. For comparison, we define an *ideal* system whose performance is unaffected by the response of an *ideal* subsystem.

Definition 6.1.1 *Ideal Subsystem: A subsystem which offers* zero *delay to service a request is called an* ideal *subsystem. The response of this subsystem is called an* ideal *response.*

Definition 6.1.2 *Latency Tolerance: It is the degree to which the system performance is close to that of an* ideal *system. The* tolerance index *is its metric.*

Definition 6.1.3 *Tolerance Index (for a latency):* Tolerance index, $tol_{subsystem}$, *is the ratio of* $U_{p,subsystem}$ *in the presence of a subsystem with a* non-zero *delay to* $U_{p,ideal\ subsystem}$ *in the presence of an* ideal *subsystem. In other words,* $tol_{subsystem} = \frac{U_{p,subsystem}}{U_{p,ideal\ subsystem}}$.

As discussed in the introduction, there are two ways of defining an *ideal* subsystem: as a *zero delay* subsystem or as a *contention-less* subsystem. We prefer the former for the following reason. Consider the tolerance of network latency. Let the number of processors in a system be scaled. We believe that the performance of the *ideal* system should not change. In other words, if the network latency is tolerated, the performance of a processor should not be affected by changes in either the system size or a data placement strategy (which affects parameter values for remote access patterns). Thus, the choice of a zero-delay subsystem is amenable to analyze the *latency tolerance* for more than one subsystem at a time.

A tolerance index of *one* implies that the latency is tolerated. Thus, the system performance does not degrade from that of an *ideal* system.[1] We divide the system performance in the following zones:

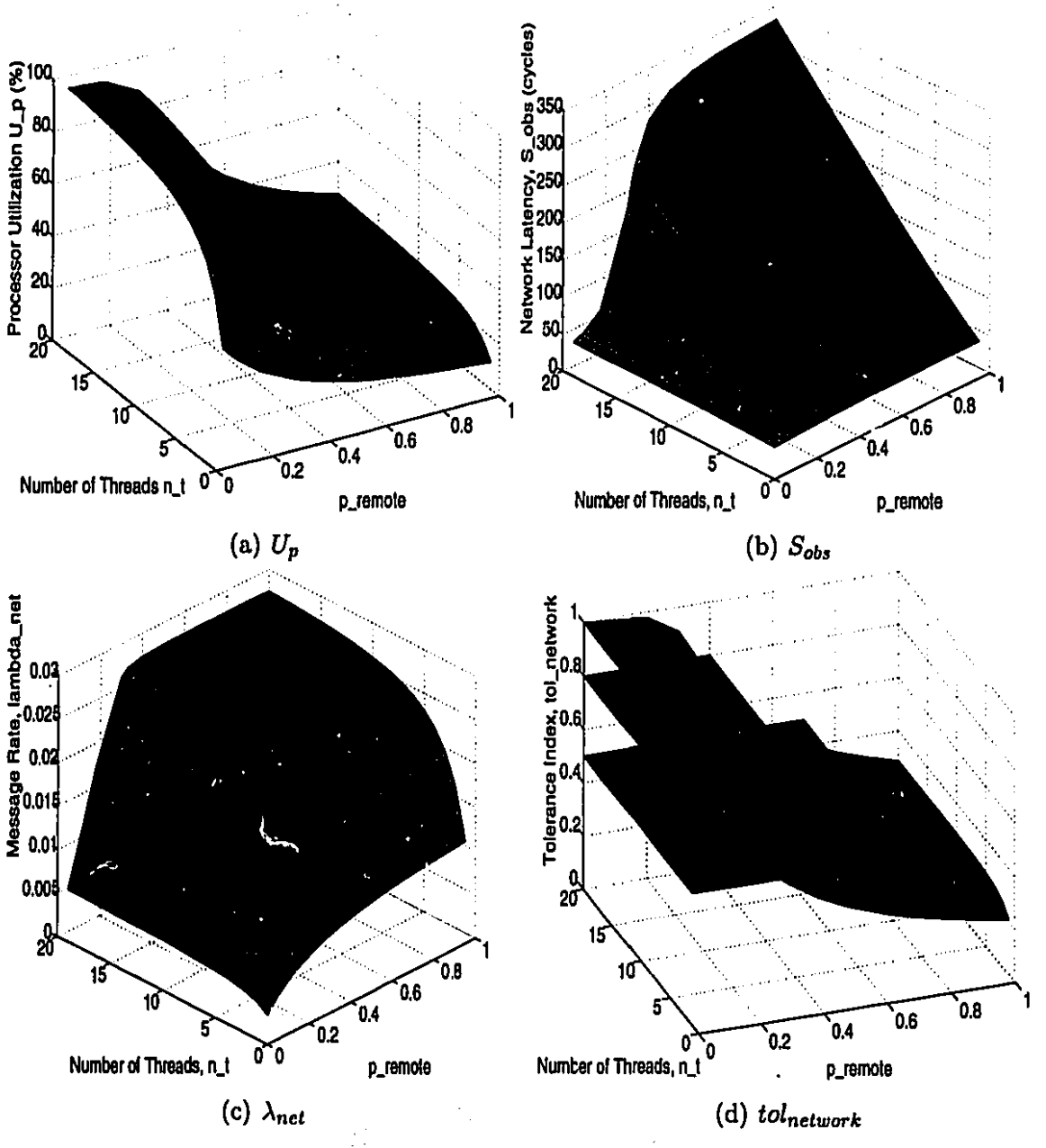
$tol_{subsystem} \geq 0.8$: the latency *is tolerated.*

$0.8 > tol_{subsystem} \geq 0.5$: the latency is *partially tolerated.*

$0.5 > tol_{subsystem}$: the latency is *not tolerated.*

The choice of 0.8 and 0.5 is somewhat arbitrary, except for the fact that the corresponding U_p values are 0.8 and 0.5, when other subsystems do not affect U_p.

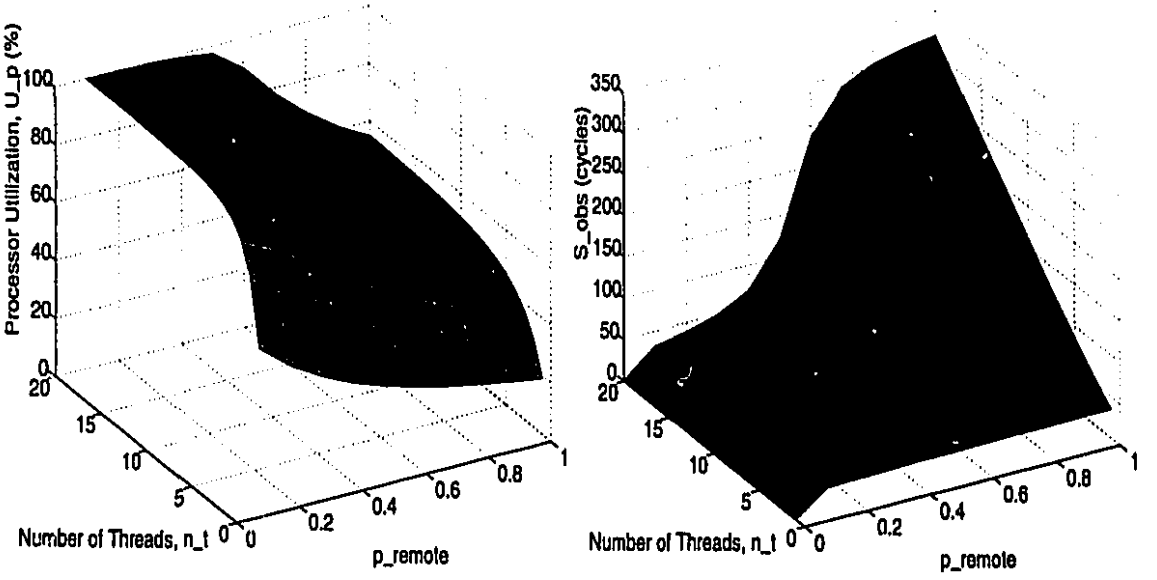
With the above background, we next analyze the network latency tolerance.



(a) U_p (b) S_{obs}

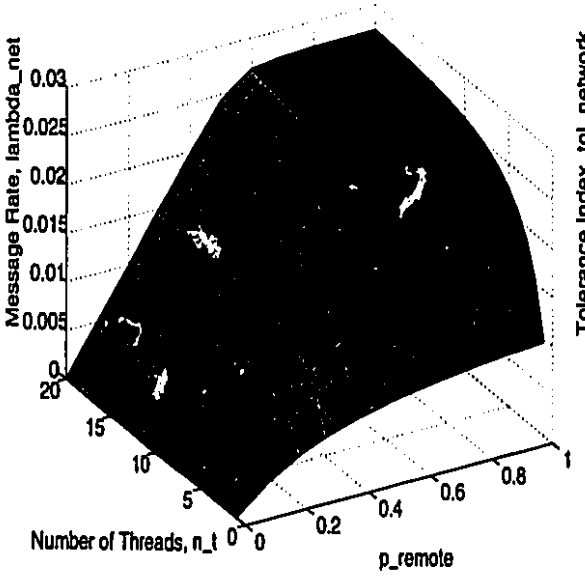
(c) λ_{net} (d) $tol_{network}$

Figure 6.1: Effect of Workload Parameters at $R = 10$.

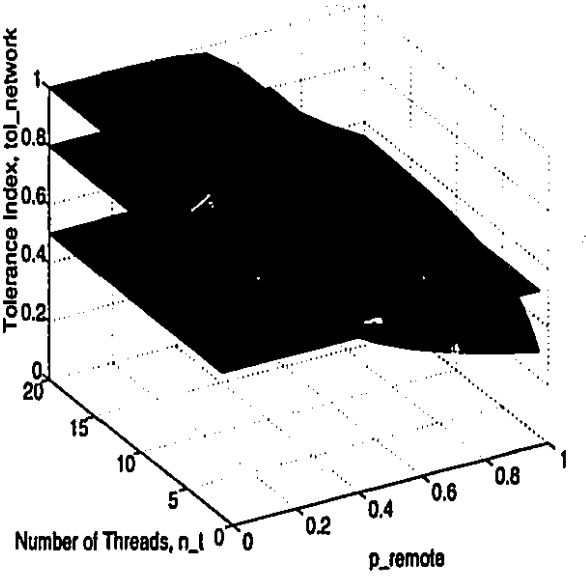


(a) U_p.

(b) S_{obs}.



(c) λ_{net}.



(d) $tol_{network}$.

Figure 6.2: Effect of Workload Parameters at $R = 20$.

6.2 Network Latency Tolerance

In this section, we show the impact of workload parameters on network latency tolerance. We study the performance of processor and IN subsystems simultaneously, with the following purpose: *For what workload characteristics is the network latency tolerated?*

Figures 6.1 and 6.2 show U_p, S_{obs}, λ_{net} and $tol_{network}$ for R= 10 and 20, respectively. The placement of U_p and $tol_{network}$ plots adjacent to S_{obs} and λ_{net} highlights the effect of workload parameters on both subsystems, the processor and the network. We analyzed the U_p, S_{obs} and λ_{net} with these model parameters in Chapter 5. In this section, we will review the performance of network and processor subsystems, study the behavior of $tol_{network}$, and analyze the impact of a thread partitioning strategy on the latency tolerance.

Network Performance:

Figures 6.1(c) and 6.2(c) show that the network is a bottleneck for high values of p_{remote}. So, λ_{net} saturates at 0.029. On saturation, the IN routes the maximum number of messages per unit time per processor under given remote access pattern. Each message travels d_{avg} hops on the IN on average, and so does its response. Thus analytically, a maximum rate at which messages return to a processor from the IN is:

$$\lambda_{net,saturation} = \frac{1}{2\ d_{avg}\ S} \quad (= 0.029,\ \text{for } p_{sw} = 0.5 \text{ and } S = 10.) \tag{6.1}$$

$\lambda_{net,saturation}$ is independent of workload characteristics (except the remote access pattern). Figures 6.1(c) and 6.2(c) show that λ_{net} saturates at $p_{remote} = 0.3$ and 0.6, respectively.

When λ_{net} *saturates*, the network latency varies as follows (see Figures 6.1(b) and 6.2(b)):

1. For a fixed value of n_t, S_{obs} remains constant (at a high value) with respect to p_{remote}. The network is a bottleneck, so the number of messages on the IN (either waiting or being routed) becomes a constant.

2. If p_{remote} is constant and n_t increases, then more messages wait on the IN. So, a linear increase in S_{obs} occurs with n_t.[2]

[1] Section 6.4 shows an exceptional case when the performance may exceed that of the ideal system.

[2] If the switches on the IN have limited buffering, then S_{obs} will saturate with n_t. We do not investigate the effect of buffering on IN switches, in this thesis.

When λ_{net} is *below saturation*, we observe the following:

1. When n_t is constant, an increase in p_{remote} diverts more messages to remote memories. So, λ_{net} increases linearly with p_{remote}. Since the contention on the network increases with p_{remote}, S_{obs} starts from an unloaded value and rises rapidly at high p_{remote}.

2. For a p_{remote}, λ_{net} increases with n_t and almost saturates by $n_t = 5$. S_{obs} increases linearly with n_t, and the rate of increase is small at low p_{remote} values.

Processor Performance:

Let us trace the U_p values at $n_t = 4$ to observe the effect of p_{remote}, in Figure 6.1(a). U_p is close to 100% for $p_{remote} \approx 0$. In other words, the processor receives a response to (one of) its accesses before it runs out of work. An increase in p_{remote} increases S_{obs}, and beyond a critical p_{remote}, U_p decreases, according to Figures 6.1(b) and 6.1(a) respectively. At this critical p_{remote}, a remote access travels $2d_{avg}$ hops for a round trip on the IN and spends $2S$ time units to get on/off the IN. The remaining fraction of accesses is serviced locally. Our back-of-the-envelope analysis in Chapter 5 (Equation 5.12) shows that:

$$\begin{array}{lll} \text{message rate from processor} & \leq \text{local memory service rate} & + \text{message rate from network} \\ \frac{1}{R} & \leq \frac{1-p_{remote}}{L} & + \frac{1}{2(d_{avg}+1)S} \\ p_{remote} & \leq 1 + \left(\frac{L}{2(d_{avg}+1)S} - \frac{L}{R}\right) & \end{array} \tag{6.2}$$

For $R = 10$ and 20, the above equality occurs at $p_{remote} = 0.18$ and 0.68 respectively. From Figure 6.1(a), we observe that U_p drops for $p_{remote} \geq 0.18$, because the remote accesses take longer time to return. Let us consider three zones for U_p based on p_{remote} values of 0.18 and 0.3 (the value at which IN saturates):

$p_{remote} \leq 0.18$: In this region, the processor does not run out of work, and U_p is high. Since S_{obs} and λ_{net} are small, U_p is unaffected by network delays.

$0.18 \leq p_{remote} \leq 0.3$: U_p drops with an increase in the value of p_{remote}, because a rapid rise in S_{obs} increases the delay for remote accesses. At large n_t, U_p remains high, despite a high value of S_{obs}.

$0.3 \leq p_{remote}$: In this region, the IN becomes a bottleneck i.e. λ_{net} saturates. Saturation value of U_p is low, but does not change in spite of an increasing value of n_t (and S_{obs}).

Given an n_t, an increase in p_{remote} increases the number of threads waiting for a remote response. U_p decreases with increasing p_{remote}, in spite of a constant S_{obs}, because the fraction of threads serviced locally diminishes.

Figure 6.2 shows a similar behavior. The corresponding values of p_{remote} are higher because R is higher.

Figure 6.1(a) and (d) show that a use of 5 to 8 threads results in most of the performance gains. For the MMS, on average, each class of threads has three functional units (a processor, a memory and a switch). When service times are balanced, 3 threads (or accesses) on 3 units are serviced. The remaining 2 to 5 threads (or accesses) in the waiting queue at each unit help to tolerate the differences in service times and their distributions.

Tolerance Index:

While the absolute value of U_p is critical to achieve a high performance, the tolerance index signifies whether the latency of a subsystem is a performance bottleneck. To compute $tol_{network}$, there are two ways to analytically obtain the performance of an *ideal* system:

- Modify system parameters: Let the switches on the IN have *zero* delays, then the performance can be computed without altering the remote access pattern. The disadvantage is that this method is not useful for measurements of an existing system.
- Modify application parameters: Let p_{remote} be *zero*, then the ideal performance for an SPMD-like model of computation is computed without the effect of the network latency. The disadvantage is that the remote access pattern needs to be altered. We prefer this method, since it is applicable to existing systems.

Figures 6.1(d) and 6.2(d) show the tolerance index ($\frac{U_p}{U_{p,ideal\ network}}$) for the network latency at $R = 10$ and 20, respectively. Horizontal planes at $tol_{network} = 0.8$ and 0.5 divide the processor performance in three regions: S_{obs} is *tolerated* ($tol_{network} \geq 0.8$); S_{obs} is *partially tolerated* ($0.8 > tol_{network} \geq 0.5$); and S_{obs} is *not tolerated* ($0.5 > tol_{network}$).

Recall from Equation 6.2 that when p_{remote} is less than the critical value, the rate of memory accesses (with maximum value of $\frac{1}{R}$) is less than the throughput of memory and IN

subsystems (i.e. memory and IN bandwidths). On average, a processor receives a response before it runs out of work. Figure 6.1 shows that even at a small n_t (5), $tol_{network}$ is as high as 0.86. When the IN becomes a bottleneck at $p_{remote} = 0.3$, $tol_{network}$ drops to 0.75. A higher value of R tolerates a p_{remote} value as high as 0.6 (see Figure 6.2).

Both figures show that $tol_{network}$ is low when the IN saturates. For an unsaturated IN, $tol_{network}$ is higher. An obvious question is: *Does S_{obs} determine $tol_{network}$?* The following example shows that S_{obs} does not determine $tol_{network}$. We focus on particular performance points from Figures 6.1 and 6.2, which have similar S_{obs} values (as shown in Table 6.2). At $R = 10$, note that $n_t = 8$ tolerates an S_{obs} of 53 time units, but $n_t = 3$ does not. Similarly at $R = 20$, $n_t = 6$ tolerates an S_{obs} of 56 time units, but n_t=3 and 4 only partially tolerate S_{obs}. So, *what determines the region of operation?* For the same architectural parameters, different combinations of n_t, R and p_{remote} can yield the same S_{obs} but different $tol_{network}$. A combination of low p_{remote} and either a high n_t or a high R, exposes (and performs) more work locally in the PE, (for example, p_{remote}= 0.2, n_t = 8, and R=20), and hence $tol_{network}$ value is higher. Thus, for an unsaturated IN, the following ways can improve $tol_{network}$:

1. A low p_{remote} reduces the number of messages on the IN, resulting in a lower S_{obs} and higher $tol_{network}$. The disadvantage is that the messages are diverted to local memory module for service, thereby increasing its response time L_{obs}. For a special case of a small n_t ($\approx$ 1), the network traffic is also low, hence U_p is close to the performance of an *ideal* IN, and $tol_{network}$ is close to 1.

2. An increase in R (from 10 to 20, in the example) reduces the number of messages to IN and local memory. Thus, S_{obs} and L_{obs} decrease and $tol_{network}$ increases. Note from Figure 6.2(d) that S_{obs} is partially tolerated for p_{remote} as high as 1.0.

3. Lastly, an increase in n_t increases $tol_{network}$ due to availability of more work (a higher n_t with same R indicates that more iterations of a loop are exposed/forked at a time). However, the disadvantage is a significant increase in response times at the switches (collectively S_{obs}) and at the local memory module (L_{obs}).[3]

We note the following points for the network latency tolerance:

[3]Agarwal [4] reports a deteriorating effect of partitioning of a cache at a large n_t. Thekkath et al. [91] and Eickemeyer et al. [35] report little variations in cache miss rates due to multithreading. In this thesis, we do not explore this application-dependent phenomenon.

1. Workload characteristics and not the resulting S_{obs} value determine whether the network latency is *tolerated, partially tolerated* or *not tolerated.*

2. For a set of values for architecture and workload parameters, there exists a critical p_{remote} beyond which the network latency cannot be tolerated.

3. Increase in R improves $tol_{network}$, and also increases the critical value of p_{remote} up to which the network latency is tolerated.

R	n_t	p_{remote}	L_{obs}	S_{obs}	λ_{net}	U_p	$tol_{network}$
10	8	0.2	40.7	52.7	0.0164	81.94	0.929
	4	0.3	20.6	53.4	0.0170	56.80	0.710
	3	0.5	14.8	54.7	0.0177	35.45	0.473
20	6	0.4	17.0	56.1	0.0175	87.55	0.899
	4	0.5	14.9	55.2	0.0175	70.18	0.741
	3	0.7	13.1	53.6	0.0174	49.69	0.543

Table 6.2: Network Latency Tolerance, with $R = 10$ and $R = 20$.

Impact of a Thread Partitioning Strategy on Latency Tolerance

Performance objectives of a thread partitioning strategy are to minimize communication overheads and to maximize the exposed parallelism [84, 18]. Recall from Section 5.2 that our model assumes the threads as iterations of a `doall` loop. So, performance related questions are: *How many iterations should be grouped into each thread? And, how do the workload parameters affect the tolerance?*

Let us assume that our thread partitioning strategy varies n_t and maintains the exposed computation constant (at a time), by adjusting their R values, i.e. $n_t \times R$ is constant.[4] Figure 6.3 shows $tol_{network}$ with respect to n_t and R. Horizontal planes at $tol_{network} = 0.5$ and 0.8 divide the $tol_{network}$ plot in three regions:, where S_{obs} is tolerated, partially

[4]This is similar to the grouping of accesses by Boothe [18] to improve R. For large grouping the message size will affect the routing delay, S on a switch. Here, we will ignore this effect.

tolerated, and not tolerated. We highlight certain values of $n_t \times R$ from Figure 6.3 in Table 6.3 and Figure 6.4. For $n_t \times R = 40$, Table 6.3 shows that:

1. A low p_{remote} results in higher $tol_{network}$, because a smaller fraction of memory accesses wait for S_{obs}.

2. At a fixed value of p_{remote} (say, 0.2), $tol_{network}$ is fairly constant, because U_p and $U_{p,ideal\ network}$ increase in almost the same proportion with R.

3. For $R \leq L(= 10)$, L_{obs} is relatively high and degrades U_p values. Since $U_{p,ideal\ network}$ is also affected, $tol_{network}$ is surprisingly high.

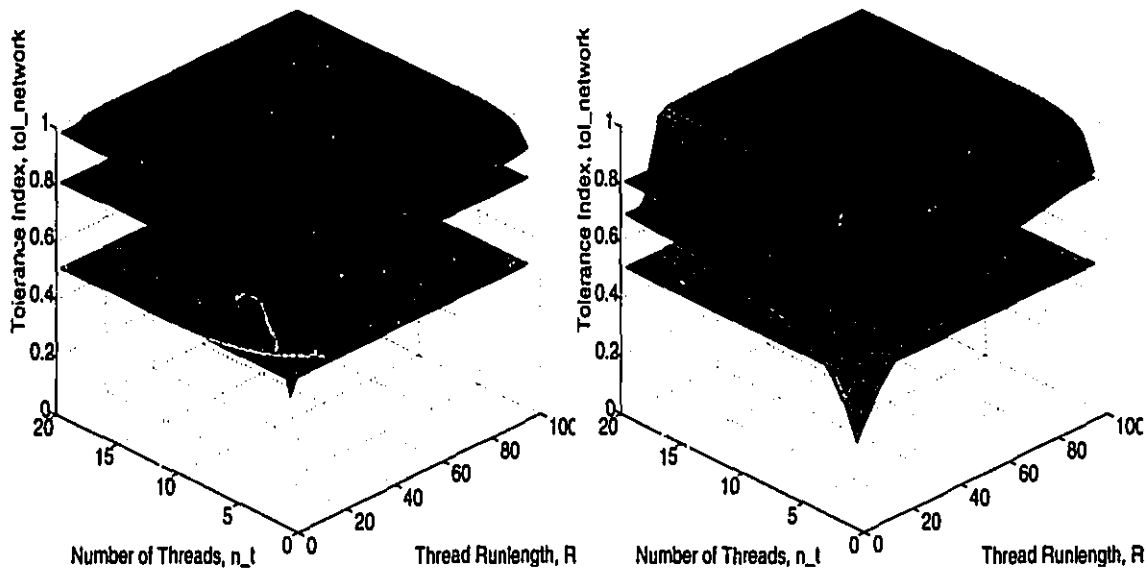
When $R \leq L$, Figure 6.4 shows a convergence of $n_t \times R$ lines, because the memory subsystem has more effect on $tol_{network}$ (as discussed in next section). We note that:

1. For $R \geq L$, the $tol_{network}$ (and U_p) value is close to maximum at $n_t = 2$. Thus, a high R achieves good results, but n_t should be more than 1.

2. A high value of $n_t \times R$ exposes more computation at a time, so $tol_{network}$ is high.

p_{remote}	n_t	R	L_{obs}	S_{obs}	λ_{net}	U_p	$tol_{network}$
0.2	2	20	13.1	33.2	0.0069	69.74	0.825
	4	10	22.7	43.5	0.0133	66.29	0.829
	5	8	30.0	48.6	0.0153	61.02	0.843
	7	6	47.1	55.7	0.0173	51.96	0.891
0.4	2	20	12.9	38.4	0.0111	55.45	0.656
	4	10	17.6	61.3	0.0190	47.64	0.596
	5	8	19.7	75.0	0.0212	42.49	0.587
	7	6	22.8	103.7	0.0237	35.60	0.610

Table 6.3: Effect of Thread Partitioning Strategy on Network Latency Tolerance.



(a) $p_{remote} = 0.20$. (b) $p_{remote} = 0.40$.

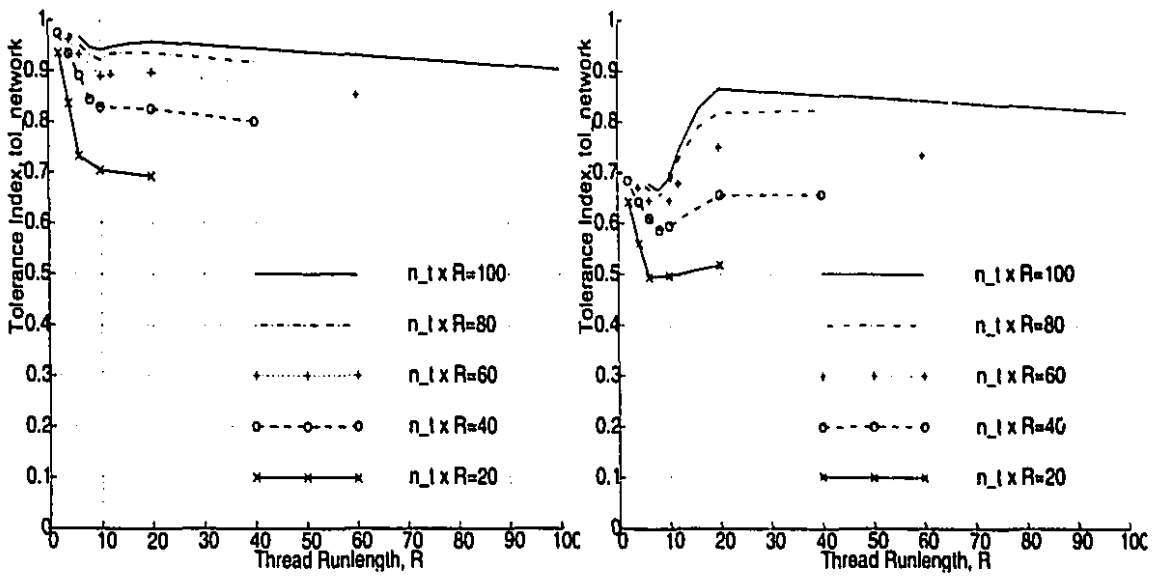
Figure 6.3: $tol_{network}$ with two p_{remote} values.

6.3 Memory Latency Tolerance

In this section, we discuss the tolerance of memory latency using workload parameters. Figure 6.5 shows tol_{memory} for two values of L, when $p_{remote} = 0.2$. Horizontal planes at $tol_{memory} = 0.5$ and 0.8 divide the tol_{memory} plot in three regions: L_{obs} is tolerated, partially tolerated, and not tolerated. For $R \geq 20$ and $n_t \geq 6$, tol_{memory} saturates at 1.0, i.e., L_{obs} does not affect the processor performance. Table 6.4 focuses on sample points for which $n_t \times R$ is constant. Note that the data for $L = 10$ is same as that for $p_{remote} = 0.2$ in Table 6.3. The differences in tol_{memory} and $tol_{network}$ from the two tables indicate that:

1. A high $tol_{subsystem}$ does not necessarily mean a high U_p, *unless* the latencies of all subsystems are tolerated. (When $R \geq L$, U_p is proportional to $tol_{memory} \times tol_{network}$.) Thus, a low $tol_{subsystem}$ indicates that the *subsystem* is a performance bottleneck.

2. The impact of n_t on L_{obs} is significant at low p_{remote}, because more messages are diverted to local memory. For a change in n_t from 2 to 7, L_{obs} increases by 3-folds. Table 6.3 shows a smaller change in L_{obs} at high p_{remote}.



(a) $p_{remote} = 0.20$. (b) $p_{remote} = 0.40$.

Figure 6.4: Network Latency Tolerance for Thread Partitioning Strategy.

For $R \leq L$, memory subsystem dominates the performance. Table 6.4 shows that:

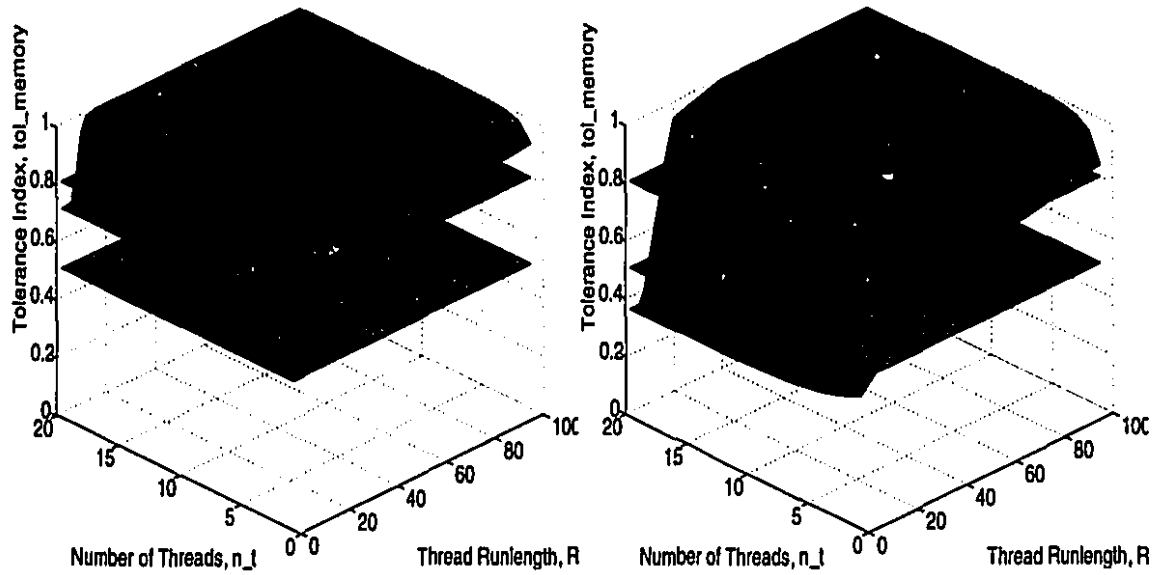
1. An increase in L from 10 to 20 increases L_{obs} by over 2.5 times. Also, tol_{memory} is at most *partially tolerated.*

2. $R \geq L$ results in high tol_{memory} and U_p, because each thread keeps the processor busy for longer duration. A side effect is a lower contention at the memory.

   For the thread partitioning strategy (which keeps $n_t \times R$ = constant), a high R value means low n_t, and further reduces contentions. The result is a high tol_{memory}.

Similar to the observation on the network latency tolerance, we note that depending on the workload characteristics, the same value of L_{obs} can result, when the MMS is operating in any of three tolerance regions.

In Section 6.2 on the network latency tolerance, we mentioned that performance gains beyond 5-8 threads were negligible (as also reported by others [100, 4, 90]). We conjectured that this was due to exhaustion of hardware parallelism (per processor). To verify the conjecture, we focus on a single node of the MMS. For $R = 2$ and $L = 10$, Figure 6.6



(a) $L = 10$ (b) $L = 20$

Figure 6.5: tol_{memory} with respect to workload parameters.

shows the effect of number of ports at the memory on U_p. A low R highlights the effect of hardware parallelism. For $n_t > n_p$, a linear increase in U_p with n_p. Also, note that U_p saturates with increasing n_t, when $n_t > n_p$. In other words, once the memory ports are busy, the processor performance cannot be improved using n_t. Thus, the processor performance improves with n_t in the presence of a higher hardware parallelism (n_p).

6.4 Scaling the System Size

The scaling of the system size raises the following questions for a compiler to optimize the workload parameters: *How will the tolerance of network latency change with the system size? Which parameters have significant effect on the tolerance?* First, we discuss the effect of distributions for a remote access pattern, i.e. *geometric* and *uniform* (where a remote access is directed to any of the memory modules with equal probability). We show that the effect of locality on the latency tolerance is significant. Second, we study the performance of subsystems and show that a careful tuning of the workload can exploit the IN for a better performance than an *ideal* (very fast) IN.

L	n_t	R	L_{obs}	S_{obs}	U_p	tol_{memory}
10	2	20	13.1	33.2	69.74	0.843
	4	10	22.7	43.5	66.29	0.797
	5	8	30.0	48.6	61.02	0.763
	7	6	47.1	55.7	51.96	0.729
20	2	20	32.3	31.7	55.01	0.665
	4	10	67.2	35.4	41.62	0.501
	5	8	87.5	36.2	35.28	0.441
	7	6	128.0	37.0	27.82	0.390

Table 6.4: Effect of Thread Partitioning Strategy on Memory Latency Tolerance, when p_{remote} is 0.2.

Figure 6.7 shows $tol_{network}$ when the number of processors, P, is varied from 4 to 100 (i.e. k=2 to 10 processors per dimension). At $p_{remote} = 0.2$, n_t is varied for two runlengths. We observe that:

1. For a *uniform* distribution, d_{avg} increases rapidly (from 1.3 to 5.0) with the system size, and the network latency is not tolerated. $tol_{network}$ saturates with low n_t and high k. But for a *geometric* distribution, d_{avg} asymptotically approaches $\frac{1}{1-p_{sw}}$ (= 2) with increase in P, and an increase in n_t improves $tol_{network}$ close to 1. The performance for the two distributions coincides at $k = 2$ for all n_t values.

2. For all the machine sizes, $tol_{network}$ is close to its saturation value for 5 to 8 threads. Note that even a large system does not require a large n_t to tolerate network latency.

3. At $R = 10$, and k from 6 to 10, $tol_{network}$ increases up to 1.05 for a geometric distribution, i.e. the system performs better than with an *ideal* IN. The delays at network switches alleviate the contentions at remote memories, thereby improving the response for local accesses.

4. An increase of R increases $tol_{network}$ values and the maximum $tol_{network}$ value is close to 1. A higher R reduces the memory access rate. A reduced contention at the memory decreases L_{obs}. This improves U_p as well as $U_{p,ideal\ network}$.

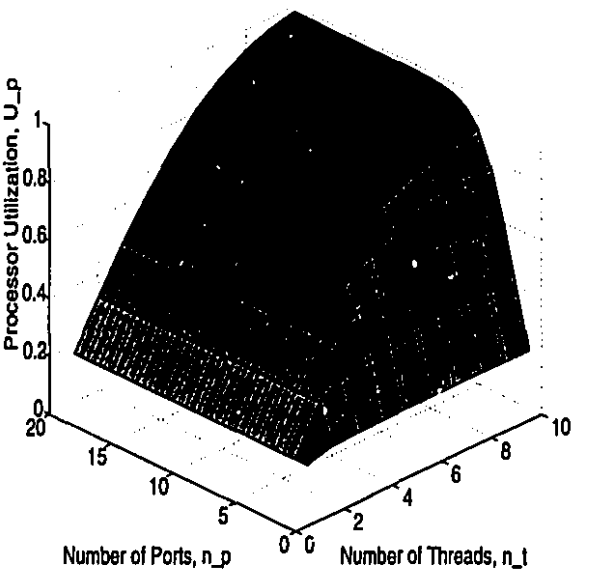
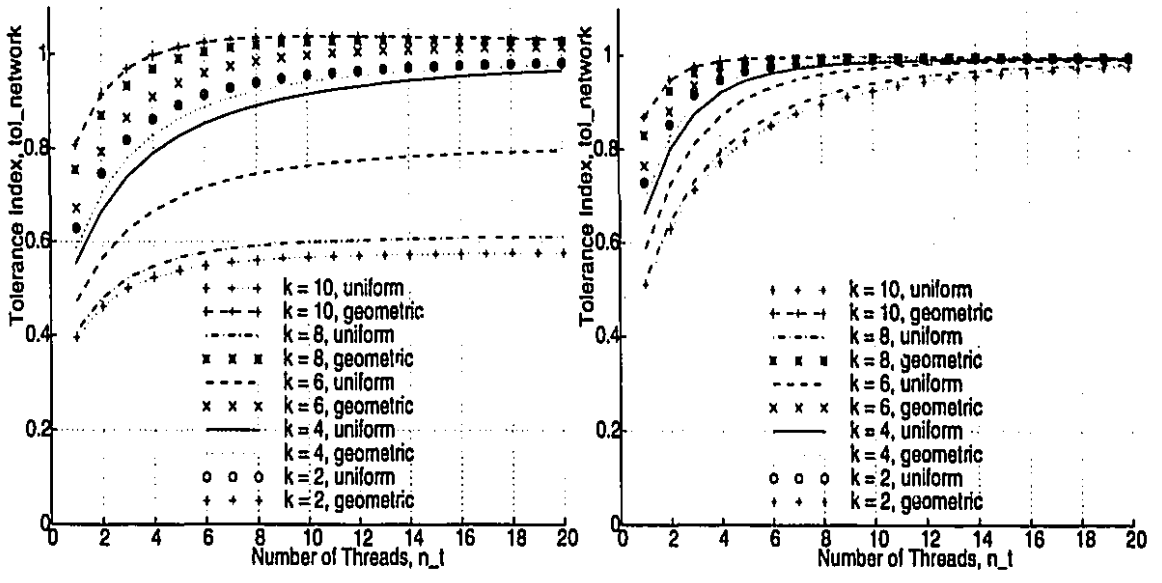


Figure 6.6: U_p with respect to number of memory ports n_p.

Now, we focus on the observation 3 stated above. Figure 6.8 shows that with increasing the system size, the system throughput ($= P \times U_p$) increases, when $n_t = 8$ and $R = 10$. For the *uniform* distribution, the network latency increases rapidly and the throughput is low. In contrast, a *geometrically* distributed access pattern shows an almost linear increase in throughput (slightly better than the system with an *ideal* IN). Transit delay for all remote accesses on an *ideal* IN is *zero*. Accesses from all processors contend at a memory module increasing the L_{obs} (see Figure 6.8(b)). Thus, $U_{p,ideal\ network}$ is affected. For a *geometric* distribution, the IN delays the remote accesses at each switch (similar to the stages in a pipeline), just enough to result in a low S_{obs} and L_{obs}. The local memory accesses are serviced faster, and U_p values improve. The following are the two implications from the above observations:

- A very fast IN may increase the contention at local memory, and the performance suffers, if memory response time is not low. Multiporting/pipelining the memory can be of help. Also, prioritizing the local memory requests can improve the performance of a system with a very fast IN.



(a) $R = 10$. (b) $R = 20$.

Figure 6.7: Tolerance Index for different system sizes.

- A larger system can make better use of IN under a good locality to tolerate network latencies than a smaller system.

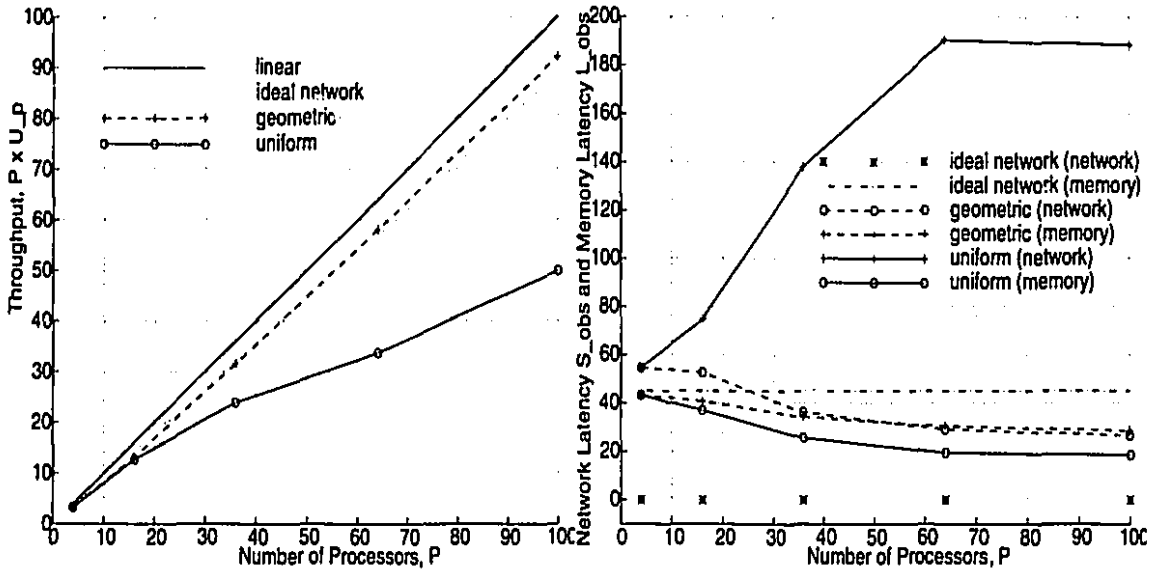
6.5 Discussion

In summary, our results on the latency tolerance show that:

1. The extent of the latency tolerance depends on the choice of workload parameter values rather than the resulting value of latency. A large latency does not necessarily degrade the system performance.

2. The network latency is tolerated *only if* the memory access rate is less than the rate at which memory and IN can respond. We compute the critical p_{remote} value up to which network latency is tolerated.

3. A high U_p requires high tolerance indices for both the network and memory latencies. A low $tol_{subsystem}$ indicates that the subsystem is a performance bottleneck.



(a) Throughput. (b) S_{obs} and L_{obs}.

Figure 6.8: System throughput for uniform and geometric remote access pattern.

4. A high thread runlength yields a high U_p, and tolerates the latencies better than a high n_t. Under our assumption of a small number of ports for a realistic memory subsystem, performance gains for n_t beyond 5 to 8 threads are negligible because hardware parallelism in the system is exhausted.

5. A large system with a good locality in remote access pattern can make good use of the IN as a pipelined buffer and relieve contentions at the memory. So, the throughput increases *almost* linearly with the system size and is up to 5% better than with an *ideal* IN. The use of a very fast IN leads to an increased contention at the memory. Hence, the performance suffers.

Item 5 suggests that to improve the performance we should also prioritize the requests from local processor helps to keep the local computation unaffected by contentions. This approach has been adopted in the processing node design of the EM-4 multiprocessor system [83].

6.6 Related Work

The latency avoidance techniques, like caches and memory hierarchy, continue to be studied extensively in the literature [86, 7, 73, 78]. The effectiveness of latency tolerance techniques, like multithreading and prefetching, has not been studied formally in a way suggested in this chapter. Kurihara et al [54] evaluate the effectiveness of multithreading using up to two threads (since their applications did not have more inherent parallelism). Their cost analysis shows a reduction in remote memory access costs with the use of two threads. They report a simultaneous increase in the network latency and channel utilization. Those conclusions are in conformity with our analytical results. Further, we have defined the latency tolerance and applied it to analyze the performance bottlenecks in the system.

Analytical performance evaluation studies by Agarwal [4] and Saavedara-Barrera et al. [80], modeled a multithreaded processor in a cache-based multiprocessor system. Willick [101], Johnson [50] and Adve [2] modeled a closed system. Weber [100] and Thekkath [90] simulated bus- and network based multithreaded systems. Most of these studies focus on processor performance and report that 4 to 5 threads per processor yield a performance increase while higher parallelism decreases the processor throughput. However, none of these work analyze the latency treatment formally.

Our analysis provides significant information on how to tolerate a latency and the impact of a thread partitioning strategy on the latency tolerance and system performance. on latency tolerance provides additional significant information We also show that the saturation of processor performance occurs at a small number of threads, because the hardware parallelism (number of resources per processor) in the system is exhausted.

For single-threaded machines, the Flash system [42] has been studied by comparing its performance with an *ideal* machine. They use a *contention-less* network, and *zero-delay* for cache protocols. Similar to our observation (increased memory response time due to ideal IN), they report an increase in contention at the cache due to *ideal* behavior of the bus and memory. They use a fixed size system (of 16 processors) for their results. So, they do not observe the effect of unloaded network latency on the system performance. Section 6.1 shows that unloaded network latency varies with system size, and should be factored out of the performance of the *ideal* system. Section 6.4 shows how the latency tolerance changes with the system sizes.

6.7 Conclusions

In this chapter, we have introduced a new metric called the *tolerance index*, $tol_{subsystem}$, for analyzing the latency tolerance in an MMS. For a subsystem, $tol_{subsystem}$ indicates how close the performance of a system is to that of an ideal system. The interaction of subsystems in an MMS plays an important role in determining the performance. We provide an analytical framework based on closed queueing networks, to compute $tol_{subsystem}$.

Our results show how the latency can be tolerated as long as the rate at which a processor sends memory accesses is less than the rate at which the subsystems can respond. Further, the latency tolerance depends on the choice of workload parameter values and inherent delays at the subsystems, rather than the latency for individual accesses. The most performance gains result from 5 to 8 threads due to exhaustion of hardware parallelism. A pipelining or multi-porting of system resources provides increased hardware parallelism essential to exploit the high software parallelism (say, $n_t > 5$).

The latency tolerance is useful to identify the performance bottlenecks. For high performance, both the memory and network latencies have to be tolerated. Since the number of parameters in a multithreaded system is large, the latency tolerance helps to narrow the focus of performance optimizations to the parameters, which affect the performance the most. Thus, an analysis of the latency tolerance yields more insights into the performance optimizations than an analysis of processor utilization.

In the next chapter, we apply our analytical model to analyze McGill's EARTH multithreaded system. This case study shows the effect of multithreading in the presence of realistic subsystem interactions.

Chapter 7

Case Study: EARTH-MANNA System

In previous chapters, we developed analytical performance models of abstract multithreaded systems and analyzed their predictions. Our focus was on the effectiveness of multithreading to achieve high performance. We studied how the network traffic and latencies increased with multithreading, and how the network performance affected the processor utilization. Earlier in Chapter 4, we discussed how the design of a multithreaded processing node affected its performance.

A real multithreaded system, like McGill's EARTH-MANNA multiprocessor system, presents more challenges to performance modeling and analysis due to complications of realistic subsystem interactions under multithreaded program executions. This chapter focuses on the problems 3.2.1, 3.2.2, 3.2.4, and 3.2.5 discussed in Chapter 3. The objectives of this chapter are as follows. *First*, we extend our analytical performance model to analyze the EARTH-MANNA system. *Second*, we validate the model predictions using the measurements from program executions on the EARTH-MANNA system.

The EARTH (Efficient Architecture for Running Threads) architecture supports a multithreaded execution model based on split-phase communications and synchronizations [46]. Currently, the EARTH architecture is implemented on a 20-node EARTH-MANNA multiprocessor hardware testbed [59]. The EARTH-MANNA processing nodes (with Intel 860 XP) are connected across a high-bandwidth network, consisting of a hierarchy of crossbar

switches. Henceforth, we refer to the EARTH-MANNA system as the EARTH system, except where we discuss the implementation details of the MANNA system.

We apply the analytical model in Chapter 5 to predict the performance of the EARTH system. Extensions to the analytical model are two-fold. First, we develop two approximations to the mean value analysis (MVA). These approximations account for complex interactions among the resources in the EARTH system, and the characteristics of a realistic multithreaded workload. Second, we expand the set of parameters to represent the program workload in the EARTH Threaded-C. Such workload characterization is helpful for performance related optimizations.

First, on the architectural aspect, we model the *simultaneous possession* of the bus, for accesses in an EARTH node. For example, when the processor at an EARTH node is accessing the local memory, no access from remote processors to other functional units on this node can proceed. With our heuristic and the iterative nature of the MVA, we formulate the above problem under one analytical model (unlike at least two models in [48, 56]). Each request to the resources, memory or network interface, contends at the bus, and releases the bus at the completion of the access. Thus, the queueing delay for the access is the sum of service time for each queued request through the bus, rather than the queueing delay at individual resource alone.

Second, for a program workload, the thread characteristics at different processing nodes may differ. Each type of request (e.g. local or remote memory accesses) requires a different service time from the server (memory system). So, a single unified queue length alone, as used by the existing MVA [75], is not enough to compute the queueing delay for a specific request. To improve the accuracy, our heuristic to the MVA considers the service demand for each individual access in the queue at a subsystem, and the numbers and types of requests in the queue at the time when this request enters.

Inputs to our performance model are, the program workload parameters, like the number of threads, thread runlengths, the number of split-phase long latency operations, and the architectural parameters derived from the EARTH system. The model predicts the processor utilization and the latency for a remote access with split-phase operations. We characterize the variation of these performance measures with program workload parameters. The runtime measurements from the EARTH system on synthetic benchmark programs match

within 5% of the analytical model predictions in most cases. These predictions also conform well with the measurements on the real programs (reported in [59]). These results demonstrate how the reliastic costs of multithreading affects the performance of the fine-grain parallel program workload.

A split-phase multithreaded operation may involve accessing the local memory, sending messages on the network, receiving responses, and performing synchronization operations. Through the performance characterization using workload parameters, we show what are the realistic latencies experienced by individual accesses during a multithreaded program executions, and how they affect the processor performance. These latency values are significantly higher than their base values typically reported in the literature. Such characterization provides a strong evidence on how useful our analytical model is to compilers and system architects of multithreaded systems for performance related optimizations.

This chapter is arranged as follows. In Section 7.1, we describe the relevant details of the EARTH system. In Section 7.2, we outline our analytical model for the EARTH system, and develop approximations to the mean value analysis (MVA). In Section 7.3, we validate our model predictions using the runtime measurements on the EARTH system. We characterize the variation in the performance measures with respect to program workload parameters. We discuss the related work in Section 7.4, and summarize major results of this chapter in Section 7.5.

7.1 Experimental Testbed

This section contains a brief description of the McGill EARTH-MANNA system and the program workload to characterize the system.

7.1.1 EARTH Architecture

The EARTH (Efficient Architecture for Running Threads) architecture proposes that synchronization operations and computations can be efficiently performed using separate functional units [46]. A node in an EARTH multiprocessor consists of an Execution Unit (EU) to execute threads sequentially, and a Synchronization Unit (SU) to support synchronization operations in parallel program executions and communication with remote processing

nodes. Currently, the EARTH programming model is implemented on the MANNA multiprocessor, developed at GMD FIRST in Berlin, Germany [20]. The EARTH Threaded-C compiler supports multithreading primitives by expanding them inline in order to reduce their overheads. This section describes McGill's EARTH system. Appendix D reports further details.[1]

System: The EARTH multiprocessor system consists of multiple EARTH nodes across a high-bandwidth interconnection network (IN). Figure 7.1 shows a multiprocessor configuration for EARTH system. EARTH nodes are connected to the leaves of the interconnection network. Each EARTH node, as shown in Figure 7.2, consists of two Intel i860 XP RISC processors clocked at 50 MHz, 32 MB of DRAM, and a fast network interface called the link. The network is a hierarchy of 16×16 crossbar chips [20].

EARTH Node: An EARTH node has an Execution Unit (EU), a Synchronization Unit (SU), and a part of distributed shared memory (see Figure 7.2). The EU and the SU interact through *ready* and *event* queues maintained in memory at the same node. The EU executes the application program code. The SU performs the synchronization and communication operations.

EU: To start a computation, the EU fetches a *thread id* from the ready queue, and executes a thread to completion. The EU issues a long latency memory access (local or remote), places them in the event queue, and context switches to another ready thread.

SU: The SU reads incoming messages from the event queue (from local EU), and the link_in node (network messages from remote processors). In response, the SU reads/writes to local memory, sends messages, replies to messages, updates synchronization variables, and schedules threads for execution by writing their thread id's to the ready queue.

Memory: The memory at an EARTH node maintains the local data, global data, ready queue, and event queue. An access to the local memory by the EU (or SU) incurs a service time of 10 cycle, in the absence of queueing delays.

Multiple such local memory accesses are required to complete one long latency memory access. The multithreading operation `GET_SYNC` is a long latency memory access. This operation fetches a datum, and completes a synchronization to activate a thread. A similar

[1] In this thesis, we use the terms "EARTH", "EARTH-MANNA" and "MANNA" to represent "EARTH-MANNA system". In reality, "EARTH" architecture is mapped on to the "MANNA" system.

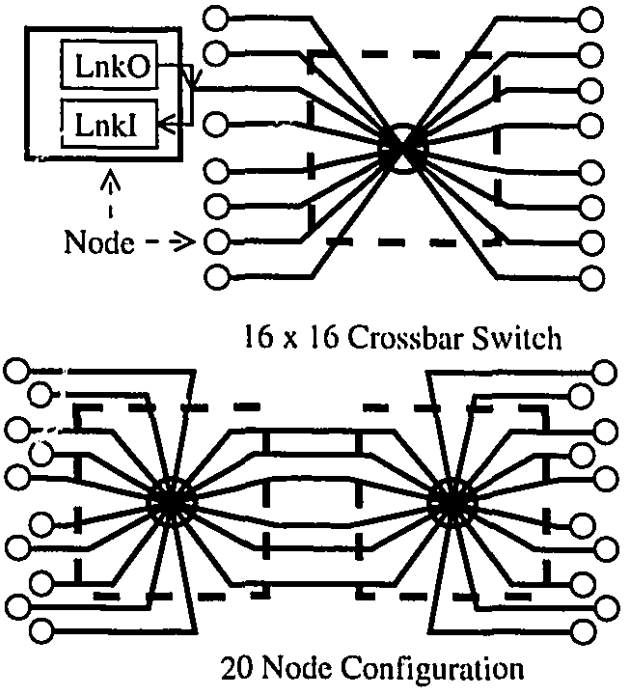


Figure 7.1: An EARTH Multiprocessor System.

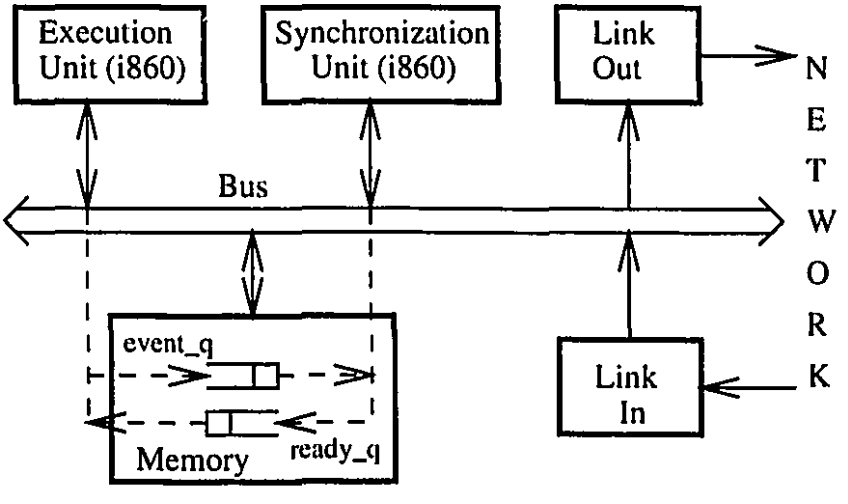


Figure 7.2: An EARTH Node.

operation DATA_SYNC is issued to stores a datum and synchronize. The EU issues these accesses to communicate among threads, which may execute on different nodes. The destination SU accesses its local memory, and routes the response to the EU originating the access.

Link Interface: The *Link_in* and *Link_out* nodes interface an EARTH node to the network. Buffers at each link node store up to 4 messages. An SU sends a message to a remote SU through the *Link_out* node, and receives a network message through the *Link_in* node.

Bus: All functional units in an EARTH node communicate through a pipelined bus, e.g., an access to a link node by the SU. The bus is held till the access to a functional unit is complete.

Crossbar: Each crossbar chip connects 16 input channels to 16 output channels in parallel. A channel is 1 byte wide, and supports pipelined transfers. Each output channel selects an input channel for message transfer in a round-robin manner. The first byte takes up to 32 cycles to reach the output channel, and thereafter, the transfers take 1 byte per cycle. After the completion of one transfer, other waiting channels, if any, are given priority for next message transfers. The interconnection network uses a hierarchy of crossbar chips. Processing nodes are connected to the leaves of the interconnection network (Figure 7.1 shows a 20-node configuration).

EARTH Threaded-C Language: The EARTH Threaded-C language is an extension to the C language. The extensions support the declaration of threaded functions, the specification of threads within these functions, and the specification of EARTH operations. The language requires an explicit specification of the partitioning of threads and the EARTH operations to be used. For example, one GET_SYNC operation fetches one remote datum, one DATA_SYNC operation stores a datum to remote location, and the END_THREAD operation performs a context switch. An explicit mention of these operations helps in obtaining the program workload parameters. We provide a synopsis of the EARTH multithreading operations in Appendix E. A sample program workload is discussed in detail, earlier in Section 2.2.

7.1.2 Program Workload

In this section we outline program workloads used to characterize the performance behavior of the EARTH system. The performance of the multiprocessor system depends on the cost of remote accesses. For a remote access, first, a request is sent across the network. Second, the access is processed at the remote node. Finally, the response is received across the network. Thus, a remote access may suffer contentions at the node, and on the network.

We use two synthetic workloads to characterize the performance. The first workload characterizes the access contentions at an EARTH node. The second workload characterizes the contentions at the node-network interface, and the interconnection network. We choose these synthetic workloads because the input parameters of interest are easier to adjust, and their effect on the system performance can be individually studied. These program workloads are written in the EARTH Threaded-C language. (Appendix E provides details on the EARTH Threaded-C language.)

Workload 1:

The objective of the first workload is to characterize the EU-SU interference on a bus at a node. That is, how much delay occurs due to contention, when a remote memory access receives a service at a particular node. We ensure that there are no messages on the network, and the contention occurs only at the node under investigation, referred as a *test node*.

A remote access reaches the test node through the link interface and the SU. To process the access at the test node, the SU fetches the requested location from the memory, prepares a response, and sends the response through the link interface. For each bus access at the test node, the SU may experience a contention from the EU. Figure 7.3 shows the program workload for the EU at the test node. The EU writes one double-precision floating point number in each iteration. There are $\frac{\texttt{Large}}{8}$ iterations. The duration between writes is controlled using the number of nops in the body of the loop. In addition, to eliminate the effect of cache on the number of write accesses during the measurements, we write to every 8th element.

The program workload ensures that the EU execution has three phases as shown in Figure 7.3. The *execute*, *read/write* and *idle* phases at an EU indicate an execution of

thread, an access by the EU waiting for the bus, and a service at the memory, respectively. For *execute* phase, the EU does not contend on the bus. "asm (''nop'');" represents the part of the computation, which does not require a memory access (see Figure 7.3). At the end of the execute phase, a write operation is issued and the EU requests an access to its local memory. The shaded *read/write* phase indicates that this access experiences a contention from the SU. The *idle* period indicates that the memory access by EU is in service, and the EU is waiting for the memory response. An access from the SU to the local memory follows the same sequence of operations as shown in Figure 7.3. We monitor the contention for the bus by varying the thread runlength.

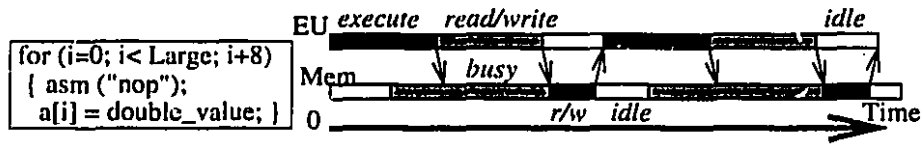


Figure 7.3: Workload for the node characterization.

Workload 2:

The objective of our second workload is to characterize the performance of the network and its interface to the node. That is, how much contention a remote memory access suffers at the network and remote processing nodes under multithreaded program execution. We want to vary the following program characteristics: the number of threads, their thread runlengths, and the number of multithreading operations for each thread.

Figure 7.4 shows the code segment for one thread of our program workload. Figure 7.4 also shows an abstraction of the program execution when two threads are active at an EU. We presented a detailed description of such program workload in Figure 2.2 (Section 2.2).

The program computes a vector addition: `a + b = c`. We chose this program workload for the following two reasons: First, it is simple to vary the program workload characteristics. Second, cache effects can be easily eliminated by a simple addressing scheme. For the purpose of our analysis, all EUs follow the same execution behavior. Each EU fetches two arrays (`a` and `b`), computes their vector sum, and stores the result (in array `c`). The code segment shows one thread. For each thread, two `GET_SYNC` operations are initiated to fetch `j`-th elements. Figure 7.4 shows that the EU at node 1 sends two messages to SU1

i.e. its local SU. The SU1 sends two remote accesses to the SU2, the SU at node 2. The SU2 responds back with the data. When the j-th elements arrive, thread_1 is triggered. After the computation, a DATA_SYNC operation stores the result in j-th element of array, c. Thread runlength is controlled using nop instructions. The EU1 sends the result and two fresh messages to the SU1. These messages are forwarded to the SU2. Thus, the program execution on a thread continues. Multiple such threads are forked at each node (as shown earlier in Figure 2.2 and Figure 2.4).

In Section 7.3, we will use the measurements from the EARTH system on these workloads to validate our model predictions. Further, our workload in Figure 7.4 provides us the flexibility of characterizing the effect of various multithreaded workload parameters on the system performance.

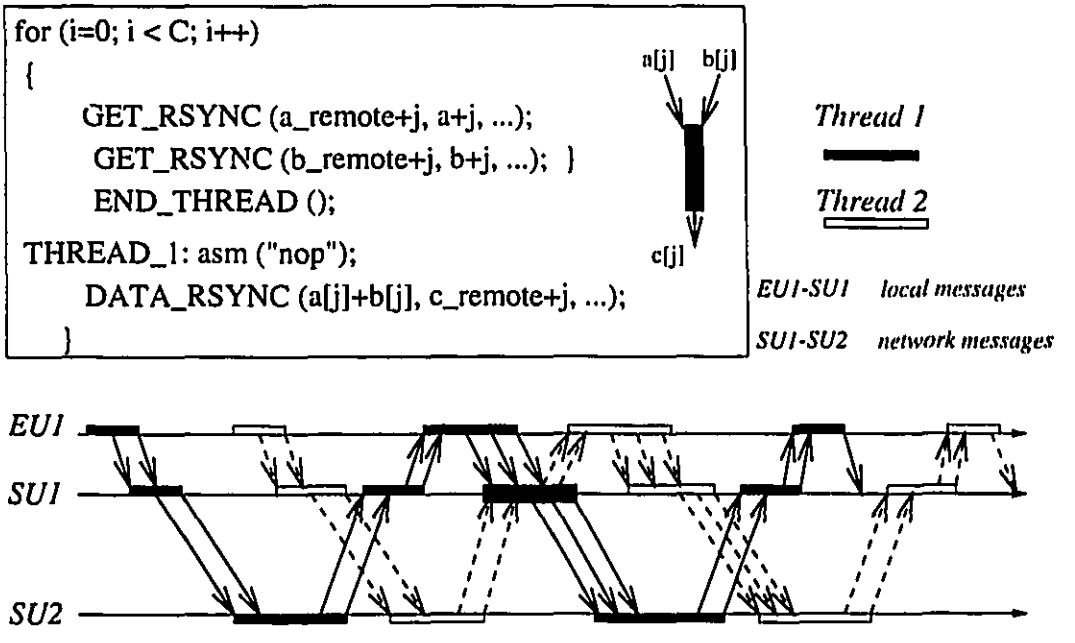


Figure 7.4: Workload with Multithreaded Operations

7.2 Analytical Model

In this section, we outline our analytical performance model for the EARTH system. This model includes the extensions to our original model (in [63]) to realistically capture the

multithreaded program executions on the EARTH system. First, we discuss the models for the functional units at an EARTH node and the network. Second, we explain the heuristics for the solution technique to predict the performance of the EARTH system. Finally, we show how to derive the performance measures of interest.

7.2.1 The Model and Its Assumptions

Our performance model of the EARTH system shown in Figure 7.5 is based on closed queueing networks (CQN). Nodes in the CQN model represent the functional units in the EARTH system and edges represent their interaction through messages. We discuss the model in more detail, below. Table B.1 in Appendix B summarizes all symbols and their experimental values for system parameters.

Our program execution model, and one sample application program are described earlier in Section 2.2. The application program is a set of partially ordered threads. THe only difference with our previously described model assumptions is that a thread is a sequence of computation and local accesses followed by *one or more* long latency accesses. A thread repeatedly goes through the following sequence of states, an *execution* at the processor, a *suspension* after issuing long latency memory accesses, and *ready* for the execution after the arrival of all responses. Threads interact through explicit long latency accesses. We assume that the application program exhibits similar behavior at each node (like a Single-Program-Multiple-Data, SPMD, model) [43]. With this assumption, one set of parameters characterizes the workload on all nodes excluding the dedicated node in the system, so the number of input parameters is reduced.

We now describe the assumptions in the closed queueing network model shown in Figure 7.5 for the EARTH system.

- All nodes in the performance model are single servers, with a First Come First Served (FCFS) discipline. Their service times are exponentially distributed. Table B.1 lists the mean value of service time for each visit to a node.
- **EU** [2]: Each EU executes a set of n_t threads, e.g., n_t iterations of a `for all` loop are forked on each node (see Section 2.2). The mean value of the service time of

[2]We use the notation "EU" and "processor" interchangeably.

a thread is R cycles. The number of local memory accesses performed during the thread runlength R. is rw, i.e. the local read/write accesses. For the context switch, C cycles are spent. We assume that threads do not migrate, so threads at a node i belong to a class i in the CQN model.

- **SU**: A multithreading operation requires one or more visits to an SU. For each visit, on average, SU_{serv} cycles are spent. This assumption is valid on the EARTH system, because the SU executes a fixed, optimized set of events of each request.

- **Memory Node**: The memory node has a mean service time of L cycles for each local access. A long latency memory access from an EU is sent to a remote memory with a probability p_{remote}. Therefore, $(1 - p_{remote})$ is the probability of a local long latency access. For requests from a thread at processor i to memory at node j, $em_{i,j}$ denotes the *visit ratio*. The value of $em_{i,j}$ depends on the distribution of remote memory accesses across the memory modules.

- **Link Nodes**: The *Link_out* node has a mean service time of $lnkout_{serv}$ cycles for each SU access. Similarly, the *Link_in* node has a mean service time of $lnkin_{serv}$ cycles for each SU access. We assume an infinite buffer capacity at link nodes for the following reason. Access time of link nodes is very fast (15 to 20 cycles per access) compared to the processing time (60 to 80 cycles per operation) at SU. Similarly on the network side, a message encounters a round-robin selection at each crossbar switch as well as the contention from other messages. Each selection at the crossbar switches incurs around 17 cycles. So, the link nodes will not exhaust their buffer spaces.

- **Bus**: An access to a resource through the bus incurs a delay of bus_{serv} (=1) cycle at the bus, apart from the delay at the resource.

- **Crossbar Switch Nodes**: Each input port of the crossbar switch node has a mean service time of xin_{serv} cycles. Each output port has a mean service time of $xout_{serv}$ cycles. Each output channel performs a round-robin to select the input channel, so $xout_{serv}$ (= 32) cycles are needed to transfer the first byte to output channel, after which the transfer is pipelined. Since the data transfer between link interface and input port is smooth and one byte is transferred at every cycle, we assign the length of the message to the service time of input port, i.e. xin_{serv} (=8) cycles.

The above models describe the behavior of individual functional units in the EARTH system. In the next section, we show how to model the interactions among functional units, based on the above behavioral descriptions of the CQN model.

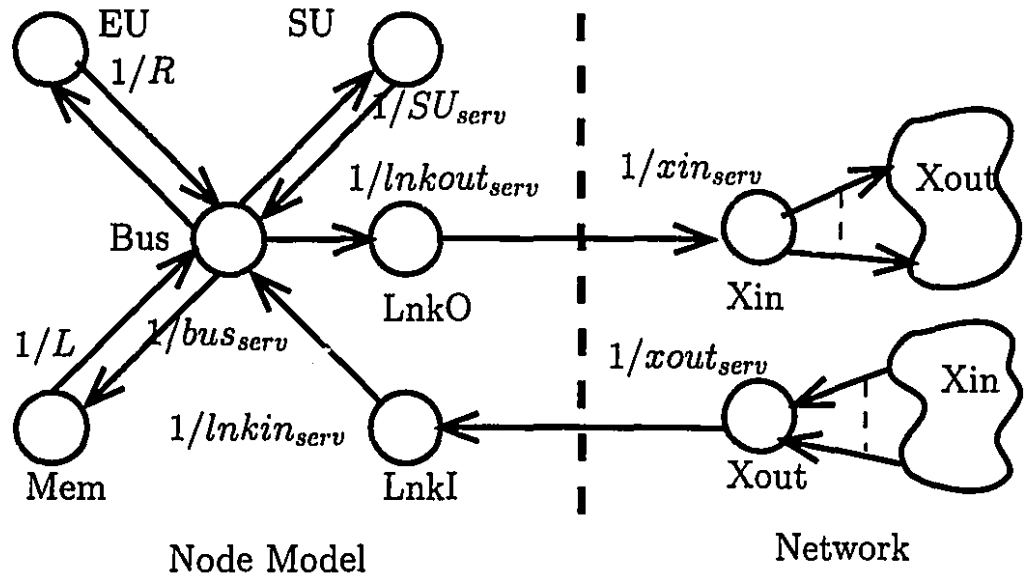


Figure 7.5: Queueing Network Model of the EARTH System.

7.2.2 Solution Technique

Our solution technique uses approximate mean value analysis (AMVA) [75]. As mentioned in Chapter 5, two salient features of AMVA are, its computational efficiency of AMVA to solve models of large systems, and its amenability to heuristics. To capture the realistic subsystem interactions of the EARTH system, we need to exploit the second feature of the AMVA. We propose two simple heuristics to predict the performance of the CQN model shown in Figure 7.5. The first heuristic accounts for the multithreaded program workload in the EARTH system, and the second heuristic accounts for thread accesses in an EARTH node, which hold the bus till the completion of their service.

The AMVA algorithm is outlined in Figure A.1 of Appendix A. With n_t threads on each processor in the system, for each class i of threads and at each node m, the AMVA computes:

- the rate λ_i at which the processor i sends memory accesses;
- the waiting time $w^*_{i,m}$; and
- the queue length $n^*_{i,m}$.

Now, we discuss heuristics to this AMVA algorithm, for performance prediction of the EARTH system.

Heuristic for Multiple Classes:

We briefly outline the heuristic to account for different service demands for accesses from different classes (processors). The later experiments show one example on the use of this heuristics: the application (program workload-2 in Figure 7.4) is executed on the test nodes and the measurement thread is executed by the dedicated node. To accurately predict the performance, such workload characteristics need to be accounted. An essential idea of the AMVA is that the queue length seen by an arriving access from class i at a node m is equal to the time averaged queuelength at the node, with one less thread in the system [75]. In AMVA (see Figure A.1 in Appendix A), Step 2(a) provides the queue length seen by this access. We note that the AMVA ignores the effect of differences in service demands for different class of accesses on the waiting time. Thus, Step 2(b) may incorrectly predict the waiting time. Our heuristic computes the waiting time $\rho_{i,m}$ for a class i access at a node m using an accurate composition of all queued requests:

$$w^*_{i,m}(N) = \rho_{i,m}\left[1 + \left(\frac{N_i-1}{N_i} n^*_{i,m}(N)\right) + \sum_{j=1, j\neq i}^{P} \frac{\rho_{j,m}}{\rho_{i,m}} n^*_{j,m}(N)\right] \quad (7.1)$$

In Equation 7.1, the service demand $\rho_{i,m}$ is multiplied by the *effective* queue length. Terms in the square bracket represent the *effective* queue length for a newly arrived access. "1" represents the newly arrived access. The second term is the queue length of class i accesses at node m. A linear interpolation is used to obtain the queue length of class i accesses before the arrival of new access in class i. The third term includes the actual queue length of class j accesses, and scales this queue length of class j accesses using their service times. In effect, the third term times the service demand represents the actual queueing delay due to class j accesses, which are already in the queue at node m.

Similar corrections to the AMVA have been considered in literature by Leutenegger [58] and others. We independently derived this approach. Our contribution is that we have successfully modeled characteristics of a multithreaded program workload with variations in service time for different multithreading operations, and we have validated the effect of this heuristic on the performance prediction using runtime measurements from a real system.

Heuristics for the Node Model:

The CQN model shown in Figure 7.5, does not have a *product-form* solution, because the following two subsystem interactions do not satisfy the assumption on the *single resource possession*, the assumption 8(a) in Appendix A:

1. When an EU accesses the memory, the bus is held till the access is complete. So, the SU cannot access other resources like link-in or link-out nodes. This subsystem interaction is an example of a *simultaneous resource possession.*

2. Crossbar switches allow a pipelined transfer of data, thus a message may be in service at more than one resources simultaneously.

Queueing delays during the above interactions are affected by the queues at more than one resources, so the *product-form* solution is not applicable [15, 75].

Now we show how to incorporate the effect of the *simultaneous resource possession* for performance prediction of the EARTH system, under one analytical model (unlike at least two models in [48, 56]). The problem of simultaneous resource possession in an EARTH node occurs as follows. Consider a local memory request issued by an EU (see Figure 7.3). When this access is in service at the memory module, three functional units (queueing nodes)— processor, bus and memory— are *simultaneously* busy to service this access. Specifically, the bus is possessed as long as the memory is accessed. An access from the SU to the local memory or link nodes has to wait till the local memory access from EU is complete. So, a new memory access does not get the bus, till the previously issued requests are serviced. A similar wait period is encountered for memory accesses by the EU, if the SU is accessing a link node. Thus, a bus request observes the waiting queue at all of the resources.

An outline of conventional approaches to this simultaneous resource possession problem is as follows [48, 56]:

1. *First*, identify a set of primary resources where an access is serviced, e.g., a memory. Identify a set of *secondary* resources, e.g., a bus, which are needed to complete the service at primary resources.

2. *Second*, isolate each such occurrence of the simultaneous resource possession for an independent model; i.e. develop one model for following functional units on each EARTH node— bus, memory, SU, link_in and link_out.

3. *Third*, derive a *flow-equivalent* queueing server for above model of functional units. That is, given a number of accesses in service, obtain the rate of their service (throughput) and the completion time for service of each access. Let us call these flow-equivalent models to be the *srp* models.

4. *Fourth*, develop a model of rest of the system, and incorporate a delay server in place of the above flow-equivalent queueing server representing functional units under simultaneous resource possession problem. Let us call this model as the *srp-free* model.

5. *Fifth*, solve the srp-free model in the third step for the desired number of threads in the system, and obtain the throughput and delay of the delay server.

6. *Sixth*, use the throughput and delay of the delay server as an input of srp models in Step 2, and from the srp-free model, obtain the throughput and delay for accesses.

7. *Finally*, iterate Steps 5 and 6 till throughputs and delays at the flow-equivalent server converge with respect to their values in the previous iteration.

Our approach to the simultaneous resource possession problem in the EARTH node is as follows: Each access to a resource on the bus waits till the bus has serviced all previous accesses to resources connected to it. Thus, the waiting queue for each access is not only the wait queue at the resources being accessed, but also a sum of wait queues at other resources on the bus. The service time for each access through the bus is the service time of the resource being accessed. In summary, although we model each resource as a separate node (i.e. a server and associated queue), we compute the waiting time at the bus in node m as a sum of the waiting time at all the resources-- link nodes, and the memory. In Appendix F, the procedure *Compute_AMVA_srp* shows the pseudo code to intergrate this

heuristic into the AMVA. Equation 7.2 provides the total wait time at the PE m for an access from class i. This value is used to compute the total wait time for a class i access and throughput.

$$\begin{aligned} w^*_{i,m,bus\ access} = \quad & (em_{i,m} + e_{i,m,bus} + e_{i,m,lnko} + e_{i,m,lnki}) \\ & \times[(w^*_{i,m,mem} + w^*_{i,m,bus} + w^*_{i,m,lnko} + w^*_{i,m,lnki}) \\ & -(\rho_{i,m,mem} + \rho_{i,m,bus} + \rho_{i,m,lnko} + \rho_{i,m,lnki})] \\ & +(\rho_{i,m,mem} + \rho_{i,m,bus} + \rho_{i,m,lnko} + \rho_{i,m,lnki}) \end{aligned} \tag{7.2}$$

Equation 7.2 shows how to compute the total waiting time due to simultaneous possession of the bus. Two parts of the waiting time $w^*_{i,m,bus\ access}$ are, the queueing delay (i.e. contention) at resources, and the service demand at each resource. The first bracket, "(...)", is the number of accesses to all resources connected to the bus. The second bracket, "(...)", is the waiting time for each access through the bus. The third bracket, "(...)", is the service demand at resources for each access through the bus. Together, the second and third brackets, "[...]", yield the queueing delay for each access, by removing the service demand from the total waiting time at each resource. The fourth bracket, "(...)", is the service demand for individual resources on the bus. The total waiting time $w^*_{i,m,bus\ access}$ for an access is a sum of the queueing delay for this access and the service demand at individual resources on the bus.

We need to model how the simultaneous resource possession during accesses to resources affect the thread runlength at the EU. When the EU executes on a thread, there are three phases (see Figure 7.3). *First*, the EU executes using the data in the cache, thus offers no contention to bus accesses from the SU. *Second*, the EU requests a location in the local memory. If an access from the SU is in service, the bus is not immediately available for the access by the EU. *Third*, the access from the EU is serviced by the memory. After the third phase, the EU returns to the first phase. We note that for the second and third phase, the EU idles, but the EU is not free to perform any other operation.[3] We assume that this idle time does not significantly change the runlength of threads under multithreaded program

[3]For this discussion, we ignore the "posted-write" mechanism on i860 XP, which allows the processor to continue execution, as long as the write access does not affect rest of the computation.

execution. The EARTH system adopts a bus arbitration scheme for a fair sharing of the bus between the EU and the SU. So, in practice, this assumption does not cause significant error. We also note that there is no software mechanism which can measure this idle time incurred by the EU. We account for the effect on the waiting time of other resources in the EARTH node. In the case of SU, we have included the effect of bus accesses on the SU processing time (using Equation 7.2 and Step 4 in Figure A.1).

Performance Measures:

We use the above modeling assumptions to derive service demands $\rho_{i,m,fu}$ for class i accesses at the functional unit fu of the node m. With the above mentioned heuristics and the AMVA [75] (outlined in Appendix A and F), we compute for class i:

1. the rate λ_i at which the processor i sends long latency memory accesses, e.g., `GET_SYNC` messages;
2. the waiting time $w^*_{i,m,fu}$ of an access at a functional unit fu of a node m; and
3. the queue length $n^*_{i,m,fu}$ for an access from class i at a functional unit fu of a node m.

Table B.1 in Appendix B lists all symbols used here. Based on λ_i, $w_{i,m}$, service times and visit ratios, we obtain the following performance measures.

Message Rate to the Network: λ_{net} is the average message arrival rate from a processor i to the IN. Each thread issues x `GET_SYNC` messages. λ_i is the rate at which processor i sends long latency accesses. A fraction p_{remote} accesses are sent to remote memory. So, $\lambda_{net,get-sync}$ is

$$\lambda_{net,get-sync} = \lambda_i \times p_{remote} \times x = \frac{n_i \times p_{remote} \times x}{\sum_{j=1}^{M} w^*_{i,j}} \tag{7.3}$$

The denominator is the total wait time at all queueing nodes for all accesses by a thread executed by the processor i. Equation 7.2 provides the total waiting time at affected functional units of each EARTH node.

Processor Utilization: U_p, is the fraction of the time an EU is neither idling nor context switching. λ_i is the rate at which long latency accesses get serviced (and threads get

enabled). Since the EU at the node i spends R cycles on each thread, the processor (or EU) utilization U_p is:

$$U_p = \lambda_i \times R \tag{7.4}$$

Latency for a GET_SYNC operation: $L_{get-sync}$, the GET_SYNC *latency* is the time between initiating a GET_SYNC operation from an EU and receiving its response at the EU. A GET_SYNC operation consists of the following phases. The EU sends a request to the local SU, which is forwarded to the remote SU. The response is sent by the remote SU through the network. On receiving the response, the local SU unwraps the message, completes the synchronization, writes to a local memory location and enables the thread suspended on this long latency access. The EU reads the local memory location and progresses on its computation. $L_{get-sync}$ is a sum of waiting time for a GET_SYNC access on each functional unit.

$$L_{get-sync} = \left(\frac{1}{\lambda_i} - \text{residence time at the processor } i\right) \tag{7.5}$$

The first term represents the total time for a thread from the start of the execution on the processor i, the suspension for long latency operation, and then the start of another execution on the processor i. Removing the second term, the residence or waiting time at the processor i, we obtain the duration for which the thread has to wait for completion of its long latency access.

Since we measure $L_{get-sync}$ from the *dedicated* processing node P, we apply Equation 7.5 to predict the same $L_{get-sync}$. The measurement of $L_{get-sync}$ proceeds as follows. A measurement thread is forked on a *dedicated* processing node, while the rest nodes execute the application program. The dedicated node sends one sample of GET_SYNC access at a time for each runlength, $R_{measurement}$. Elapsed time till response reaches the dedicated node is $L_{get-sync}$. Analytical value of $L_{get-sync}$ is given by:

$$L_{get-sync} = \left(\frac{1}{\lambda_P} - R_{measurement} - 2\rho_{P,P,bus} - 2\rho_{P,P,mem}\right) \tag{7.6}$$

The last two terms in Equation 7.6 represent the time taken by EU to read the data from ready queue.

In Section 7.3, we validate some of our analytical results by comparing with program executions on the EARTH-MANNA system. We also use the above performance model to analyze the performance of processor and IN subsystems.

7.3 Results

In this section, we show how the contentions in the EARTH system affect the GET_SYNC latency, and the processor utilization. First, we consider the effect of contentions between accesses from the EU and the SU at an EARTH node on these performance measures. Second, we characterize the no-load behavior of the crossbar switches by empirical measurements of the latencies on the network. Third, we study the effect of contentions on EARTH nodes and the network under multithreaded workloads. We validate our model predictions using measurements from the EARTH system on synthetic programs. These synthetic programs are discussed earlier in Section 7.1.2 and Section 2.2.

7.3.1 EU-SU Interaction at a Node

As discussed in Section 7.2, the GET_SYNC latency provides a measure of the waiting time at various functional units due to contentions. To measure the effect of contentions between the accesses from the EU and SU at an EARTH node, we send test messages to the *test* node under investigation. The EU at the test node accesses its local memory every R cycles. The SU contends for the bus only when test messages arrive. With a variation in thread runlength, we monitor the bus contention.

Figure 7.6 shows the effect of thread runlength on the GET_SYNC latency. Three curves represent the latency values for the model predictions and the measurements from two 2-node EARTH systems. The solid line shows measurements from the system with a crossbar, and the dotted line shows system without a crossbar but with the link interfaces of two nodes directly connected to each other. Symbols show the latencies for $R = 7$ to 14, and 19 cycles. $R = 100$ represents close to the no-load values. Each experimental value is an average of over 15 observations. In turn, each observation is an average of over 3,000 to 10,000 samples.

The difference between the analytical results and the measurements from the system with crossbar is less than 5%. In the absence of a crossbar in the system marked "+" second system, the latency decreases by 25 to 35 cycles. From Figure 7.6, we note the following:

- With a decrease in the runlength for local accesses, the contention increases significantly. For $R \leq 10$, the memory almost continuously services the request from EU.

- The number of bus accesses required for a GET_SYNC operation determines the slope of $L_{get-sync}$ curve with respect to R. We estimate that the SU at the test node makes the following 6 bus accesses in response to a GET_SYNC request:

  The *first* access reads the status of the link_in node i.e. a message has arrived. The *second* access reads the message from the link_in node. The *third* access performs a read (or write) to the local memory. The *fourth* access reads the status of the link_out node, in preparation of sending the response. The *fifth* and *sixth* accesses write the response to the buffer at the link_out node. The above estimate assumes that a request is at the link_in node and a buffer space is available at the link_out node.

  A lower number of accesses comprising a multithreading operations will decrease the latency for that operation: less time will be spent on bus accesses, and less contention will occur at the bus. Thus, an implementation of the GET_SYNC operation strongly governs its latency value.

Maquelin et al. [59] report a no-load latency of 355 cycles (with an EU overhead of 39 cycles). Thus, without the EU overhead, the no-load latency is 315 cycles. Our model predictions and measurements in Figure 7.6 show that no-load value without the EU overhead is 306 cycles, which conforms within 10 cycles. (The reason our measurements do not include the EU overhead is that as soon as the message arrives at the *dedicated* node, the function to measure the idle time is stopped. So, we measure the latency for each message separately, and do not include the EU overhead. In contrast, Maquelin et al. report an average value obtained by sending a message to remote node, receiving its response and continuing this process over a large sample.) The heuristic for *simultaneous resource possession*, outlined in Section 7.2, improved the accuracy of performance prediction in this experiment.

7.3.2 The Unloaded Network

Now, we study changes in the GET_SYNC latency due to a round-robin selection at a crossbar switch. We perform this characterization using the measurements from the EARTH system because the details of the round-robin selection policy and internal functioning of the crossbar switches are not known to us. Figure 7.7 shows the GET_SYNC latency when

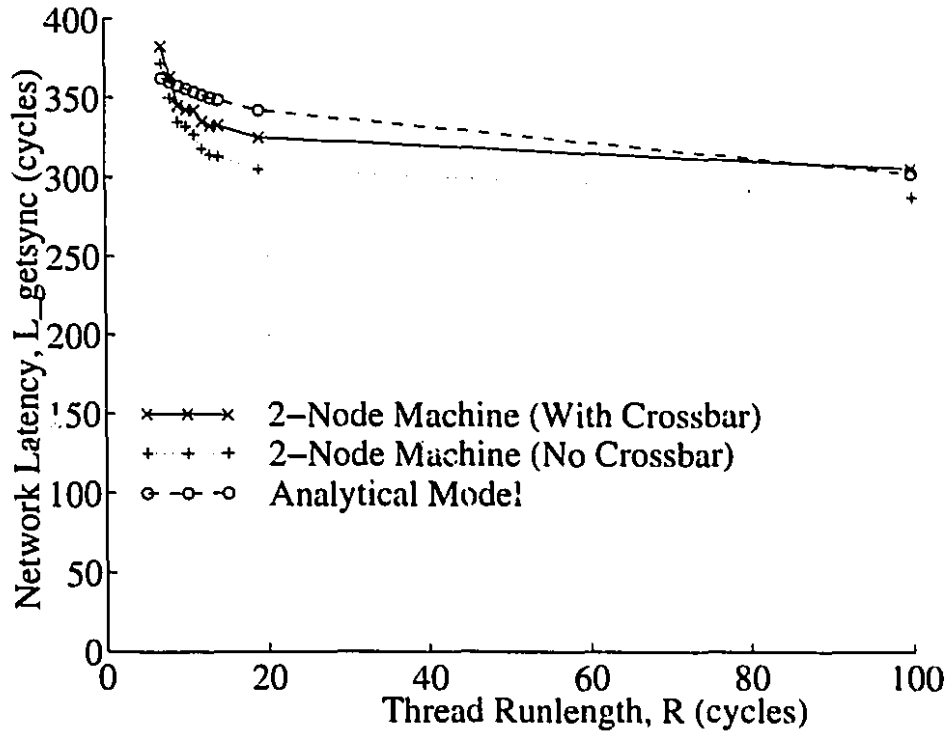


Figure 7.6: The EU-SU Contention on the Node bus

a certain number of nodes are accessed through the crossbar. In Figure 7.7, the *Same Crossbar* means that the dedicated node is on the same crossbar as the test nodes under investigation. Thus, the message from a dedicated node goes to the crossbar switch and is routed to its destination test node. The crossbar switch receives the response from the test node and routes the message to the dedicated node. The *Remote Crossbar* means that the dedicated node is on a different crossbar (see Figure 7.1). So, the message from the dedicated node is sent to the crossbar-1, say the one to which the dedicated node is directly connected. Crossbar-1 routes the message to crossbar-2, the crossbar to which the test node is connected. Crossbar-2 receives the response from the test node, and routes the message to crossbar-1, and the message is forwarded to the dedicated node. Solid lines show model predictions, and dashed lines show measurements from the EARTH system.

The program workload ensures that one `GET_SYNC` message is on the network at a time. The destination of `GET_SYNC` message is varied to characterize the effect of round-robin

selection of input channels on the delay of crossbar switches. Our analytical model assumes a stepped increase in the delay when up to 4 nodes are accessed. Specifically, the increase in delay in each direction is: 27 cycles when 2 nodes are accessed, 9 cycles for the 3rd node and 4 cycles for the 4th node. Beyond 4 nodes, the model assumes a small linear increase (of 2 cycles per node) in the delay at the crossbar. Figure 7.7 shows that the model predictions are within 5% of the experimental values for both configurations.

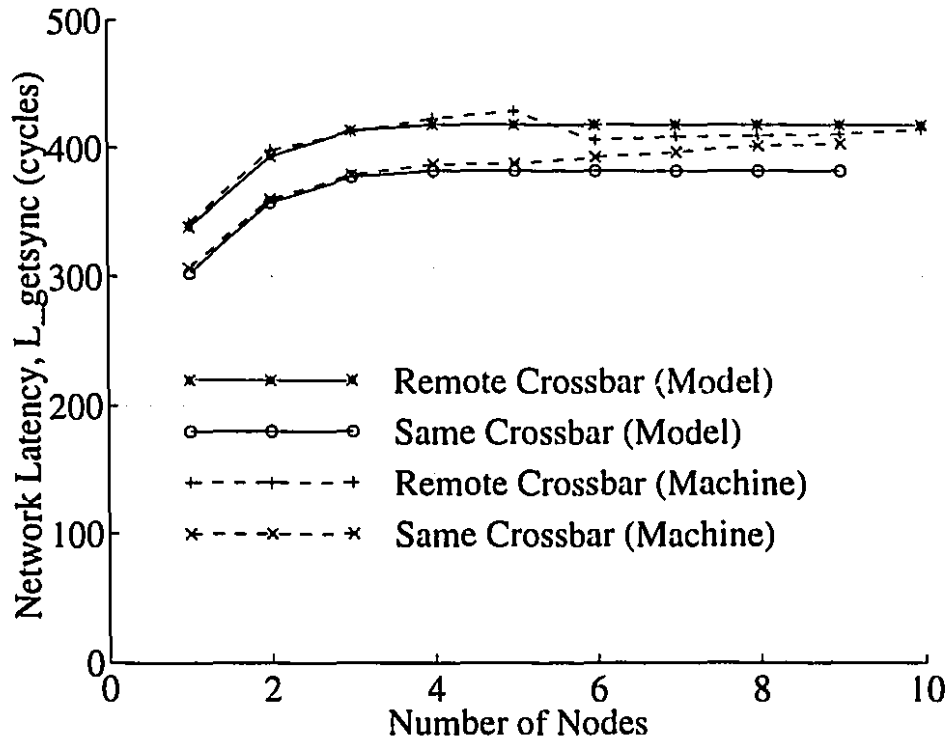


Figure 7.7: Characterizing the Crossbar Switch delay

7.3.3 GET_SYNC Latency

To study the effect of workload parameters on GET_SYNC latency, we consider the following multithreaded workload. The program performs a vector addition on remote arrays, $c[j] = a[j] + b[j]$. As explained earlier in Section 7.1.2, n_t threads are forked on each node. Each thread issues two GET_SYNC operations to fetch the *uniformly distributed* remote data.

On completion of these GET_SYNCs, the result is computed, and stored to the remote array using a DATA_SYNC operation. n_t is varied from 1 to 16. Thread runlengths are R (=1500, 3000) cycles. Parameters of the measurement thread on the dedicated node are: $n_t = 1$, $R = 200$ cycles, and one GET_SYNC operation.

Figure 7.8 shows how the GET_SYNC latency varies with n_t. Solid lines are model predictions for $R = 1500$ and 3000 cycles. Dashed lines are runtime measurements from program executions on the EARTH system. The model predictions for GET_SYNC latency conform within 5% of the measurements, in most cases. The maximum difference of 20% occurs at $R = 1500$ cycles and $n_t = 16$. One source of error is that for certain functional units we have estimated the delays which cannot be accurately measured. Specifically, these functional units are the input and output channels (buffers) of crossbar switches, link interfaces, and the SU. We can only measure the total delay for the GET_SYNC latency. We attribute the delays to various functional units based on their access times, e.g. the remote SU spends 20 cycles for each GET_SYNC access, and the remote memory spends 10 cycles. A small error in this breakdown of delays leads to a large error in prediction, particularly when the delay for a particular functional unit is large, e.g., the local SU spends up to 60 cycles to process each GET_SYNC message from the EU. From Figure 7.8, we note that:

1. At higher thread runlength, the interval between remote messages increases. A lower message rate reduces the contention at resources. Hence, the GET_SYNC latency is low.

2. The GET_SYNC latency increases with n_t, because the number of outstanding, contending messages in the system increase.

3. At higher thread runlengths, $L_{get-sync}$ saturates near no-load values when $n_t > 8$. At lower runlengths, $L_{get-sync}$ increases rapidly with n_t.

For this experiment, we notice the significance of heuristic on multithreaded workload (in Section 7.2.2). With MVA in Figure A.1, using Step 2(b) the predicted value of $L_{get-sync}$ changed by less than one cycle, even with an order of magnitude changes in n_t and R. The reason is that even for $n_t = 16$, the queue lengths at functional units (other than EUs) are very small (< 0.1). With our heuristics, in Equation 7.1, we achieved a good agreement with the measured values.

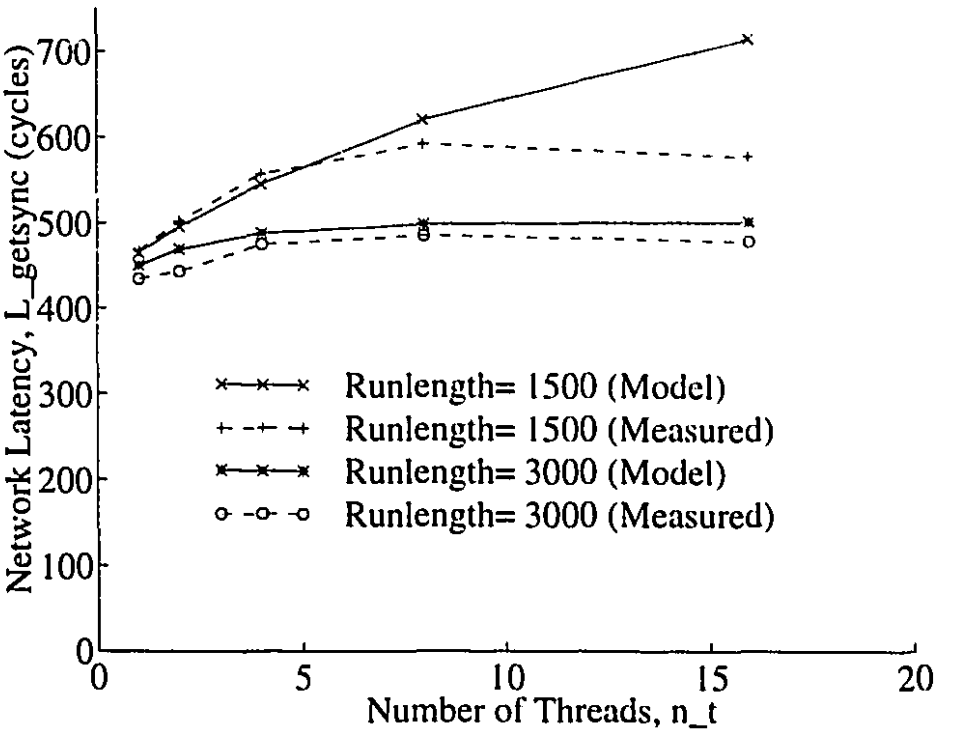


Figure 7.8: Effect of Workload Parameters on $L_{get-sync}$

7.3.4 Processor Utilization

In this section, we study the effect of a multithreaded workload on the processor utilization, U_p. For the experiment described above, Figure 7.9 shows how the number of threads and their runlengths affect the processor utilization. Solid lines are the model predictions for R = 1500 and 3000 cycles. Dashed lines for same thread runlengths are the runtime measurements from benchmark execution on the EARTH system. We observe that the model predictions match within 5% of system measurements. At $n_t = 1$, the discrepancy is up to 20%. We make a modeling assumption that a processor idles for a local memory access. Our software measurement tool cannot measure this idle time, so our measurement is an overestimate of actual utilization at the processor. Further, the idle time at the EU is measured by calling a function in the absence of ready threads. Switching to and from the function to measure the idle time incurs an overhead. Currently, the measurement function

does not record the number of times it is invoked, so a correction for this value cannot be performed. At higher number of threads, this function is called less frequently, thereby increasing the accuracy of prediction. We observe that:

1. A higher thread runlength yields a higher U_p.
2. U_p values increase with n_t. This is an essence of the multithreading technique (an increase n_t increases U_p even though $L_{get-sync}$ increases). U_p values saturate beyond $n_t = 8$. The workload is compute-bound, so U_p saturates close to $\frac{R}{R+C}$.
3. A comparison of U_p values with that at $n_t = 1$ shows a higher speedup with an increase in n_t, when R is small. For example, the predicted speedup is 1.5 at $R = 3000$ (i.e. $U_p = 92\%$ for $n_t = 8$ and $U_p = 65\%$ for $n_t = 1$). On the other hand, the speedup is 2 when $R = 1500$ (i.e. $U_p = 87\%$ for $n_t = 8$ and $U_p = 45\%$ for $n_t = 1$). However, the absolute U_p value is low at small R.

7.4 Related Work

Now, we overview the contributions of this chapter with respect to the existing literature on performance evaluation of multithreaded architectures. These work have been validated through simulations.

Queueing Network and Petri Net Models: Saavedra et al [80] report a simple validation using results of Weber's studies [100]. However computation requirements of Saavedra's models are prohibitively high for multiprocessor systems in the presence of contentions. Analytical models in [101, 8, 2, 103] are validated using simulation results from petri net models and queueing network models. None of these models include realistic subsystem interactions for multithreaded operations at a processing node.

Other Models: Agarwal [4] proposed an analytical model based on cache parameters. Johnson [50] extended this model to include the feedback effect of network on the performance. Both models are validated using simulations of Alewife system [5]. They do not model the memory subsystem. Further, they do not study the performance

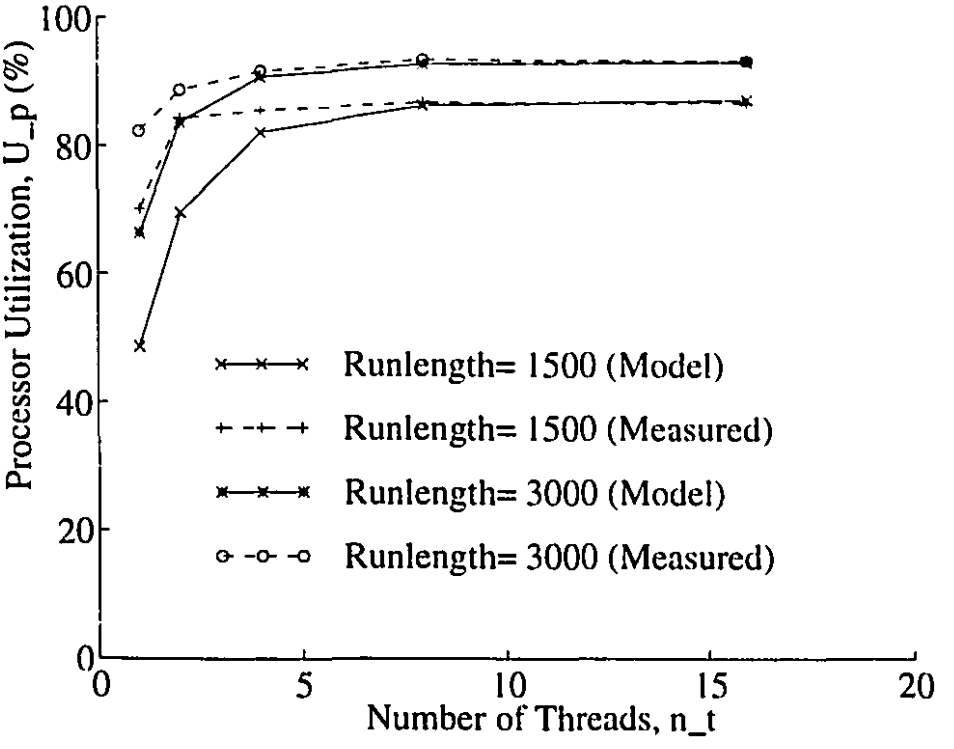


Figure 7.9: Effect of Workload Parameters on U_p

of processor and network subsystems together. So, the system bottlenecks cannot be easily located.

Simulations: Weber and Gupta [100] performed trace-driven simulations with constant context switching times, and constant shared bus latencies. Thekkath and Eggers [90] extended a similar approach using an analytical model [3] for the network performance. Waldspurger and Weihl [98] report the results of simulations on a single node of multiprocessor system. They also assume that the network is lightly loaded (no contentions).

In this chapter, we expanded the set of parameters to model realistic architectural interactions and program workload. We characterized the performance of multithreaded architectures with architectural and workload parameters. And finally, we validated our performance predictions using the measurements from the EARTH multithreaded multi-

processor system. Thus, this chapter is a significant extension over previous studies. By addressing above issues, we believe that our work has provided a strong evidence on the usefulness of the analytical models for performance optimizations on multithreaded systems.

7.5 Summary

In this chapter, we extended our analytical model (in Chapter 5) and analyzed the performance of McGill's EARTH multithreaded multiprocessor system.

We developed approximations to mean value analysis (MVA) to account for the simultaneous possession of the bus at an EARTH node, and the multithreaded workload. We showed how, given program workload and architecture parameters, to derive performance measures, like processor utilization and latency for split-phase multithreaded operations. We analyzed these performance measures using realistic costs of multithreaded operations.

Since the multithreading poses challenges to the performance measurement, we have used the following approach for performance characterization. We developed a software instrumentation to measure the latency, based on sample messages to the nodes under investigation. We validated the performance predictions using measurements on the EARTH system. Our model predictions of the `GET_SYNC` latency and processor utilization conform to within 5% of the runtime measurements on the EARTH system for synthetic benchmarks under typical program workloads. Maximum differences are around 20%, when thread runlengths are small (< 1500 cycles).

Our results indicate how the processor performance improves with increasing thread runlengths. For example, at thread runlength of 1500 cycles, with architectural parameters of the EARTH system, the multithreading improves the processor performance up to 100% of a single-threaded program execution. The `GET_SYNC` latencies increase significantly up to *two* times their base values, when R is small (1500 cycles) and n_t is high (say, 16). In Chapter 8, we will discuss the performance related optimization of a program workload, and the effect of the number of local and remote accesses per thread on the performance. We will also discuss the effect of changes in architectural parameters on the performance. Specifically, we consider two systems based on the EARTH: The first system has low access times for certain functional units at an EARTH node. The second system has delays of a

NOW system (networks-of-workstations) for the interconnection network. Finally, we will show an advantage of multithreading, in comparison to single-threaded execution, in terms of reduced sensitivity of performance to the data locality.

Chapter 8

Applications to Performance Optimizations

8.1 Introduction

In Chapters 4 and 5, we developed analytical models to predict the performance of uniprocessor and multiprocessor multithreaded systems. We analyzed multithreaded systems and showed critical values of parameters to achieve high processor performance. In Chapter 7, we applied the performance model to analyze the EARTH system. We validated our model predictions of performance measures using measurements from actual program executions.

This chapter presents a next step of a performance evaluation study. The objective of this chapter is: how to optimize the performance of a multithreaded system? Specifically, we address the following questions:

1. *What is the impact of program workload characteristics on the EARTH system performance?* In Section 8.2, we characterize the performance behavior of the Single-Program-Multiple-Data (SPMD) computation. The measurements from the EARTH system show the cost-benefits of various tree-like strategies to fork parallel computation threads on multiple nodes. In Section 8.3, we study how remote and local accesses affect the performance of the EARTH system, where the system bottlenecks are, and for what values of parameters, multithreading yields significant performance

benefits. We show that the number of long latency accesses per thread should be 3 or less, and thread runlengths should be greater than 1500 cycles to achieve higher than 80% EU utilization on the EARTH system.

2. *How would the changes in the architectural configuration affect the performance of the EARTH system?* In Section 8.4, to evaluate the architectural trade-offs of an EARTH-like system, we analyze performance of two systems based on the EARTH architectural parameters. The first system, *NOW*, assumes that the EARTH processing nodes are connected to a network with higher delays (similar to those in a *networks-of-workstation* system, *NOW*,). Our results show that for the NOW system, with multithreading the performance improves up to 200% over a single threaded program execution when thread runlengths are larger than 3000 cycles. The second system, *fast* EARTH, assumes that service times at subsystems in an EARTH processing node are reduced. That is, the SU and the link nodes are twice as fast as the current EARTH-MANNA hardware test-bed (these costs are shown in Table 8.1). We show that the performance improvement of the *fast* EARTH over the EARTH system is by 10% on single threaded executions, even at small thread runlengths. At large runlengths and with multiple threads, the performance improvement of the *fast* EARTH reduces, because the performance of the EARTH system increases with multiple threads.

3. *How robust is the performance of the EARTH system to the changes in data distributions, when multithreading is used?* In Section 8.5, we compare the sensitivity of performance on single-threaded and multithreaded program workloads to the changes in data distributions. We propose metrics to evaluate this sensitivity, analytically predict their values, and experimentally validate the results. We show that the decrease in the performance of a multithreaded workload due to a non-optimal data distribution is less than the decrease in the performance of a single-threaded program workload. Intuitively, a non-optimal data distribution would increase the latencies. However, a multithreaded workload should tolerate long latencies. This ability of multithreaded workload indeed helps to reduce the performance loss. The implications of this result on compilers and programmers are as follows: On multiprocessor systems, proper choices for computation and data decomposition are crucial to achieve high performance using a single threaded program. With multithreading technique,

the same performance can be achieved (or exceeded) with similar efforts on the computation and data decomposition. On applications we studied, even a non-optimal data-distribution (requiring less efforts) yields a performance close to the performance of the best or optimal data distribution. For programs exhibiting irregular computation parallelism, the decomposition of computation and data is extremely challenging. Thus, for programmers and compilers to achieve high performance, the multithreading reduces the need to carefully craft data distributions.

We depart from our earlier chapters in the following way. We use results from both the performance model and measurements on the EARTH system for our analysis. An assessment of realistic costs of multithreading on the EARTH system requires runtime measurements from actual program executions. Specific cases are the program execution time for transient phases, which involve forking and joining of multiple threads. Since not all measurements are possible from the system, e.g., the contentions at subsystems like the SU and the network, the model predictions provide additional insights.

Next, we discuss performance related optimizations of a program workload on the EARTH system. In Section 8.3 , we investigate the effect of architectural and program workload parameters on the EARTH system. In Section 8.4, we study how the changes in the EARTH implementation affect the performance. In Section 8.5, we study the sensitivity of the performance of the EARTH system to data locality. In Section 8.6, we discuss the related work. We summarize the results of this chapter in Section 8.7.

8.2 Program Optimizations

In this section, we characterize the performance behavior of the SPMD computation on the EARTH system using model predictions and measurements.

Table 8.2 shows different sets of input parameters, the number of threads n_t and thread runlength R, when the number of EARTH nodes is 8. The performance model in Chapter 7 predicts the processor utilization U_p, the GET_SYNC latency $L_{get-sync}$, and the delay at an SU, "SU". "Measurements" indicates the runtime measurements of U_p and $L_{get-sync,ded}$, when the EARTH system executes the application program shown in Figure 7.4. U_p values indicate the processor utilization on EARTH nodes which execute the application program.

Parameter	Description	Values
Workload Parameters		
n_t	Number of threads at each processor	
R	Mean value of thread runlength	
p_{remote}	Probability of accessing a remote memory module	
rw	Number of Read/Writes in the duration R	
..	# and type of long latency operations GET_SYNC, BLKMOV	
D_P	A distribution of data on P procersing nodes	
$T_{P,n_t}(D_P)$	Program execution time with n_t threads on P nodes	
System Parameters (*Values measured on EARTH*)		
C	Context switch overhead (END_THREAD, Scheduling etc.)	37 cycles
L	Memory latency for each access	10 cycles
SU_{serv}	SU processing time for each access (other than BLKMOV)	20 cycles
$lnkin_{serv}$	Link access time for incoming network message	15 cycles
$lnkout_{serv}$	Link access time for sending network message	8 cycles
xin_{serv}	Delay at input port of network switch	8 cycles
$xout_{serv}$	Routing delay at output port of network switch	32 cycles
P	Number of nodes in the system.	2 to 16
Output Parameters		
L_{obs}	Observed memory latency (with queueing delay)	
S_{obs}	Observed network latency (individual message type)	
$L_{get-sync}$	Latency for `GET_SYNC` operation. *ded* subscript ⇒ dedicated node	
λ_{net}	Message rate from processor to the IN (message type)	
U_p	Processor (EU) utilization	

Table 8.1: Model Parameters for EARTH System.

$L_{get-sync,ded}$ indicates the latency for a GET_SYNC operation issued by the dedicated node (for runtime measurements). The no-load value of $L_{get-sync,ded}$ for an 8-node system is 402 cycles. "SU" values indicate the number of cycles to process a message at a remote synchronization unit. Since each thread issues three split-phase operations–two GET_SYNC and one DATA_SYNC –, the "SU" value at no-load is 60 cycles.

Let us consider the values in the **bold face** from Table 8.2, i.e. rows 1, 5, 9, and 13. When the input parameter thread runlength R is decreased from 3000 cycles to 1500 cycles (the rows 5 and 9), the corresponding number of threads is increased from 2 to 4. This increase in n_t leads to an increased contentions in the system. So, $L_{get-sync,ded}$ increases from 487.8 cycles to 545.3 cycles, i.e. by 12%. At R=3000 cycles, increasing n_t from 1 to 8 results in a 57% increase in analytical $L_{get-sync,ded}$ values. In addition, a higher n_t requires more overheads to fork threads and synchronize them at the end of computation. So, when R is low, then U_p values decrease even at high n_t. The system measurements conform with these predictions.

Table 8.2 also shows the latencies for a service at the SU. We note that for $R \leq 1500$ cycles, the latency at the SU is high. A direct consequence of increase in the latency at the functional units like the SU is that $L_{get-sync,ded}$ increases and U_p decreases. Most applications reported by Maquehn *et al.* [59] have thread runlengths of the order of 10,000 cycles to 25,000,000 cycles. Of these applications, the N-Queens program (in [59]) has the smallest runlengths, i.e. 700 cycles. U_p value of the N-Queen for one-node system is 72%. For the remaining 28% duration, the one-node system executes multithreading primitives. An 8-node system yields an absolute speedup of 6.4 (i.e. $U_p = 72 \times \frac{6.4}{8} = 57.6\%$). These values conform with the results in Table 8.2. In summary, our results show that EARTH system can efficiently support applications with granularity up to 1500 cycles.

Table 8.3 shows measurements of costs to fork (and synchronize) one thread on the EARTH system. We forked 1000 computation threads on the local as well as remote node, denoted by *local* and *remote* respectively. Recall the discussion on 5 types of threads shown in Figure 2.4 (Chapter 2). The row labeled *Normal S* in Table 8.3 indicates a sequential or a single-threaded execution. These are C type threads for steady-state computation at the local node. For a remote node, one A type thread is used to fork one C type thread, while the synchronization occurs through an E type thread. The *Normal P* indicates a parallel fork and synchronization of threads, i.e. a multithreaded execution. For a local node, one

B type thread forks multiple C type threads, which synchronize using D type threads. For a remote node, one A type thread forks multiple C type threads, which synchronize using E type threads. *Tree invocation* structures are best represented in Figure 2.4. *Tree S/P* indicates that one A type thread forks one B type thread on a local or remote node. In turn, the B type thread forks multiple C type threads. Synchronizations occur using D and E threads. *Tree P/S* indicates that multiple A type threads fork one C type thread on local or remote node. Synchronizations occur through E type threads. *Tree P/P* indicates that an A thread forks one B thread on each node. In turn, These B threads fork multiple C threads. Synchronizations occur through D and E threads as shown in Figure 2.4. *T-2* and *T-8* indicate tree structure is forked on a 2-node and a 8-node system, respectively.

For *tree* invocations, we focus on *remote* values. Note that invocation costs per thread for normal P and tree S/P structures are quite similar. The reason is that for a large number of C threads (1000 threads in *Tree-1)* in the P structure, most overheads are at the remote node and those threads are forked local to that node. When the number of C threads is small (10 threads in *Tree-2*), costs are slightly higher at 4097 ns per thread. These costs per thread indicate that tree invocation is especially useful when forking a large number of threads on a large system. Rows for *T-2 P/P* and *T-8 P/P* show how costs per thread are reduced by 4 times when 4 times the number of threads are forked on an 8-node system. In other words, even though more number of threads are forked on an 8-node system, almost no increase is observed in the program execution time.

Consider an 8-node system. We want to fork 16 threads per node. The two scenarios we consider are, the *normal P* structure, and the *tree S/P* structure. Their costs per thread are, 2798ns and 4097ns respectively. A normal P structure would require 2798ns $\times$ 16 *threads* $\times$ 8 *nodes* = 358 μs for forks and synchronizations. In contrast, a tree invocation requires (2798ns $\times$ 8 *nodes*) + (16 *threads* $\times$ 4097ns) = 88 μs, a reduction by 4.07 times. Further, a use of *T-8 P/P* structure requires only 53.9 μs (= 421ns $\times$ 16 *threads* $\times$ 8 *nodes*). This is a reason why we choose a tree invocation when a large number of threads need to be forked on remote nodes. Since costs per thread are mentioned in Table 8.3, we note that a large n_t per node requires a large time to fork and synchronize, thereby performance benefits of multithreading are reduced.

In summary of results from Tables 8.2 and 8.3, to obtain a high processor utilization on the EARTH system:

- the number of threads should be moderately high, i.e., from 2 to 8;
- the thread runlength should be high;
- a tree invocation should be used to reduce costs of forking and synchronizing threads on multiple processing nodes.

	Input		Analytical Predictions			Measurements	
	R	n_t	SU	U_p	$L_{get-sync,ded}$	$L_{get-sync,ded}$	U_p
1	6000	1	**72.1**	**74.3**	**433.3**	**416.9**	**91.8**
2		2	74.9	87.7	440.2	421.4	93.9
3		4	76.4	93.9	443.8	434.1	95.4
4	3000	1	78.8	66.3	449.7	433.9	82.3
5		2	**86.8**	**83.7**	**487.8**	**443.0**	**88.7**
6		4	94.8	90.8	498.7	474.8	91.7
7		8	99.5	92.8	502.6	485.5	93.5
8	1500	2	98.2	69.5	495.6	501.9	84.3
9		4	**120.2**	**82.2**	**545.3**	**557.0**	**85.6**
10		8	155.2	86.3	620.7	592.5	86.8
11		16	201.3	87.3	716.5	579.1	86.9
13	750	4	132.1	32.2	571.2	787.2	65.5
14		8	**182.7**	**39.3**	**678.0**	**1016**	**72.3**
15		16	279.6	45.6	874.6	1035	72.3

Table 8.2: An Example of Workload Optimization.

8.3 Performance Characterization of the EARTH System

In this section, we present the performance characterization of multithreaded architectures. This characterization demonstrates how multithreaded program workload parameters affect the performance. To illustrate our results using realistic multithreading costs, without loss of generality, we analyze the EARTH system described in Section 7.2. The default architectural parameters are mentioned in Table 8.1. These values are obtained experimentally

Invocation	Local	Remote
	ns	*ns*
Normal S	5030	8960
Normal P	3039	2798
Tree-1 S/S	5593	5073
Tree-1 S/P	3696	4097
Tree-2 S/S	4972	5041
Tree-2 S/P	3253	3256
T-2 P/S	5590	3499
T-2 P/P	3413	1678
T-8 P/S	5590	893
T-8 P/P	3413	421

Table 8.3: Costs to fork a thread: S≡ Sequential, P≡ Parallel, Tree-1≡ 10 remote threads, Tree-2≡ 1000 remote threads; T-2≡ Tree for 2 nodes; T-8≡ Tree for 8 nodes.

from the EARTH system, in the absence of contentions. Values of workload parameters are mentioned in the description of each experiment.

We characterize the variations in the GET_SYNC latency and processor utilization U_p with architectural and program workload parameters. We also measure the performance on the EARTH system, when *test nodes* execute the program workload in Figure 2.2 (Section 2.2). The highlights of our results are as follows.

- With an increase in the number of processing nodes in the system to execute a program workload, the contention increases significantly at low thread runlengths ($\leq$ 1500 cycles), and the performance suffers. We show an interesting implication for users of systems with current superscalar processors. When thread runlengths decrease from 3000 cycles (for a scalar code) to 1500 cycles (say a sustained instruction-level parallelism, ILP, of 2 is achieved), U_p decreases to half of its value. That is, the program execution time does not change! Thus, there is a need to optimize communications in a program workload to effectively utilize the program parallelism.

- We show that multiple read/write accesses for a thread to the local memory change the GET_SYNC latency and U_p by 2% to 5%. However, multiple remote accesses (GET_SYNCs) per thread have a significant effect on the performance. Our model predictions are within 5% of the system measurements for most of the workload parameter values. The model predictions provide two additional insights. First, even though the processor utilization increases with the number of threads, the GET_SYNC latencies for the application are 2 to 6 times higher than their no-load values. Second, delays at a processing node (like the SU) are the major cause for the increased GET_SYNC latencies on the EARTH system!

- We illustrate under what architectural configurations can the current EARTH implementation yield further gains. First, we consider a system with current EARTH nodes and the network delays similar to those on a *network-of-workstations*. The multithreading yields an increase in the speedup from 2 (the current value) to 3, when thread runlength is 3000 cycles in an 8-node system. Interestingly, these thread runlengths to achieve high performance are similar to those observed for the current EARTH implementation. Second, we consider a *fast* EARTH system with the costs of the current MANNA network, and reduce the access times of functional units at a processing node by 50%. The performance of the current EARTH system increases by nearly 10% under single threaded program execution, even at low thread runlengths. The performance of the EARTH system improves with multithreading, so the improvement due to *fast* EARTH reduces at higher number for threads.

We present the results as follows. First, given a program workload, how does the performance vary when the machine size is varied. Second, for a given machine size, what program workload parameters have significant impact on the processor performance. Monitoring the performance of individual subsystems is not easy during program executions (and not available to us on the EARTH system). The analytical model predicts how the program workload parameters affect the performance of individual subsystems. Finally, we predict how the system performance would change, if the architectural implementations of the EARTH system be changed.

8.3.1 Architectural Parameters

In this section, we study how sensitive is the performance of the EARTH system on a given program workload, to the changes in machine size. We consider the program workload in Figure 2.2 with the following thread characteristics. For two thread runlengths, R = 1500 and 3000 cycles, we vary the number of threads from 1 to 8. Each thread issues 3 remote data accesses (2 GET_SYNCs and 1 DATA_SYNC). Despite varying the machine size, we keep the problem size fixed. So, the program execution time reduces with an increase in n_t as well as an increase in the number of processing nodes. This investigation helps us to evaluate the computation decomposition of a program workload on a system with multiple processing nodes.

Figure 8.1 shows how sensitive is the network latency for GET_SYNC operation, when the machine size is varied from 2 to 16 nodes. Thread runlengths remain constant at R = 3000 and 1500 cycles. When R is 3000 cycles and n_t is 1, $L_{get-sync}$ values are close to no-load values for the particular number of processing nodes in the EARTH system. Note that $L_{get-sync}$ plots overlap completely, when n_t = 4 ("o") and 8 ("—") at R = 3000 cycles. At R= 3000 cycles, $L_{get-sync}$ values rise very slowly with machine size. However at R= 1500 cycles, $L_{get-sync}$ values sharply increase beyond 8 nodes in the system. The two $L_{get-sync}$ plots which rise to almost three times the no-load values on a 16-node EARTH system are for n_t = 4 and 8, when R is 1500 cycles.

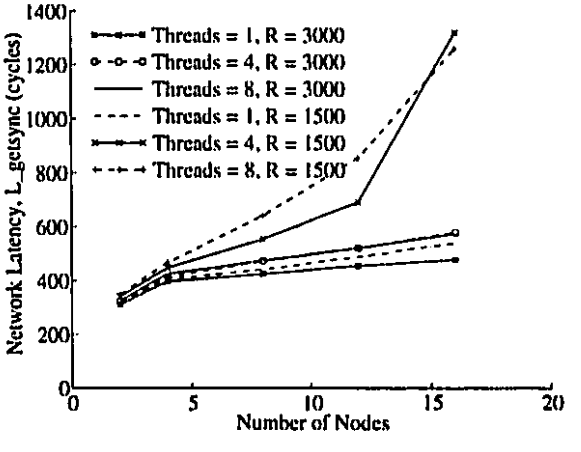


Figure 8.1: $L_{get-sync}$ characterization with number of processing nodes.

Let us consider how does the processor performance change (see Figure 8.2). As the

machine size is increased, the processor utilization consistently reduces. The change in U_p varies with R and n_t. For $R = 3000$ cycles, the decrease in U_p is small even for a 16-node system. And with 4 to 8 threads, the processor utilization is close to the saturation value. Thus, a partitioning with $R = 3000$ cycles yields a robust performance when the machine size varies from 2 to 16 nodes. In contrast, at $R = 1500$ cycles, U_p decreases rapidly with increasing machine size. Also, an increase in the number of threads (up to 8) yields little performance gains.

The following is a surprising implication from Figure 8.2 when the number of nodes is 16. With the multithreading (i.e. $n_t > 1$), U_p values at $R = 3000$ are almost double of U_p values at $R = 1500$. Each EARTH node contains Intel i860 XP processor which supports a dual-issue of instructions [20, 46]. Without loss of generality, the implication of this result on superscalar processors is as follows. Let us assume that the compiler produces a scalar code for a program workload with mean thread runlengths of 3000 cycles. For the same program workload, when compiler optimizations are applied, the generated code achieves a sustained instruction-level parallelism (ILP) of 2, i.e. mean thread runlengths are 1500 cycles. There is, however, no change in the communication characteristics. Since processing nodes place their messages on the network at a higher rate due to low R, contentions increase significantly and there is no performance gain due to increased ILP. In effect, with new techniques using a high performance processor efficiently does not guarantee a high system performance. A tuning of workload parameters as well as system architecture should pay attention to a performance analysis of various subsystems.

Thus, we note that:

- On the EARTH system, thread runlengths over 3000 cycles (i.e. 60 μs) achieves good processor utilization ($> 80\%$), even when the system contains up to 16 nodes.
- When thread runlengths are smaller than 1500 cycles, a decomposition of the program workload on to a large number of EARTH nodes causes a significant reduction in processor utilization. For these runlengths contentions at subsystems and network increase, so $L_{get-sync}$ increases and the U_p decreases.

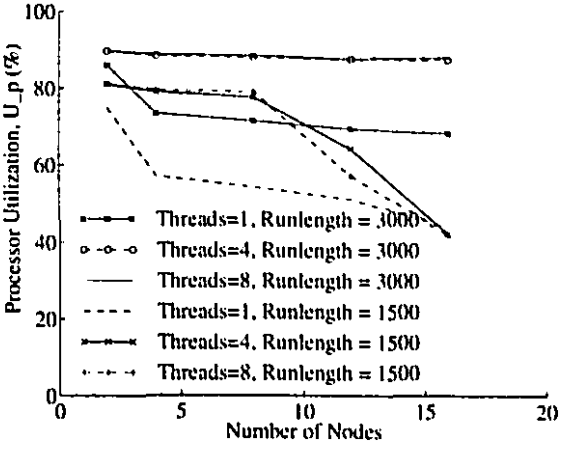


Figure 8.2: U_p for different machine sizes.

8.3.2 Multithreading Operations

We noted in Section 8.3.1 that for R above 3000 cycles, U_p is high even for machine sizes up to 16 nodes. Now, we discuss which program workload parameters affect the performance measures. We use a default machine size with 8 EARTH nodes. Thread runlengths are 1500 cycles and 3000 cycles. We vary the number of remote access operations (`GET_SYNC/DATA_SYNC`) per thread as well as the number of local read/write accesses for each thread.

Figure 8.3 shows how U_p varies with the number of threads for runlengths R= 3000 and 1500 cycles. Continuous lines are model predictions and dotted lines are runtime measurements from the EARTH system. For both runlengths, U_p values increase with number of threads and saturate. Model predictions and system measurements follow the same trends and their difference decreases at higher number of threads. Two causes of differences between model predictions and system measurements are as follows (discussed earlier in Section 7.3): *First*, we cannot measure the idle time at the EU due to its local memory accesses. *Second*, we cannot accurately account for an overhead to invoke the function for idle time measurement.

Threads with 2 `GET_SYNC` operations yield higher U_p values than threads with 3 `GET_SYNC` operations (see Figure 8.3). The difference at higher number of threads (say 8) is due to the overhead of issuing and synchronizing an extra `GET_SYNC` operation. Interestingly, U_p

values for threads with 3 GET_SYNC operations and R= 3000 cycles, are almost similar to U_p values for threads with 2 GET_SYNC operations and R = 1500 cycles. That is, if threads need more number of GET_SYNC operations, their runlengths should be high to achieve a good U_p.

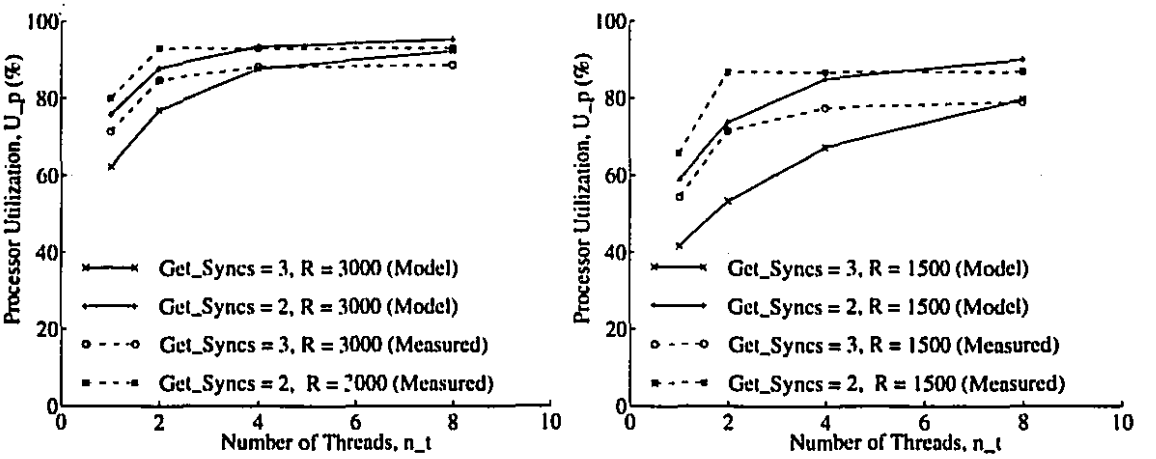


Figure 8.3: Effect of the number of remote accesses per thread on U_p.

Now, let us consider the effect of remote accesses on $L_{get-sync}$ for the two runlengths R= 3000 cycles and 1500 cycles. Figure 8.4 shows that with 2 GET_SYNC operations per thread, $L_{get-sync}$ is small and does not change significantly with other parameters, n_t and R. However, with as small as 3 GET_SYNC operations per thread, $L_{get-sync}$ increases sharply at lower R (= 1500 cycles). When n_t is 4, R is 1500 cycles and each thread has 3 GET_SYNCs, $L_{get-sync}$ values are much higher than those for the workload with n_t = 8 and 2 GET_SYNCs per thread. Despite a larger number of outstanding accesses, when n_t is 8, $L_{get-sync}$ is lower compared to the workload with n_t = 4 and 3 GET_SYNCs. Thus, the number of GET_SYNCs per thread is more crucial to the performance than the number fo threads. Curves for R = 3000 cycles show a similar behavior, with smaller differences. Notice the similarity with an observation from Figure 8.3 that $L_{get-sync}$ values for threads with 2 GET_SYNCs and runlength of 1500 cycles are almost similar to $L_{get-sync}$ for threads with 3 GET_SYNCs and runlength of 3000 cycles.

Next, we predict the effect of remote accesses on delays at the SU, one of the key functional units to support the multithreading. A runtime measurement of the delay at an SU is not possible in the software, because the SU at a remote node is only one of the

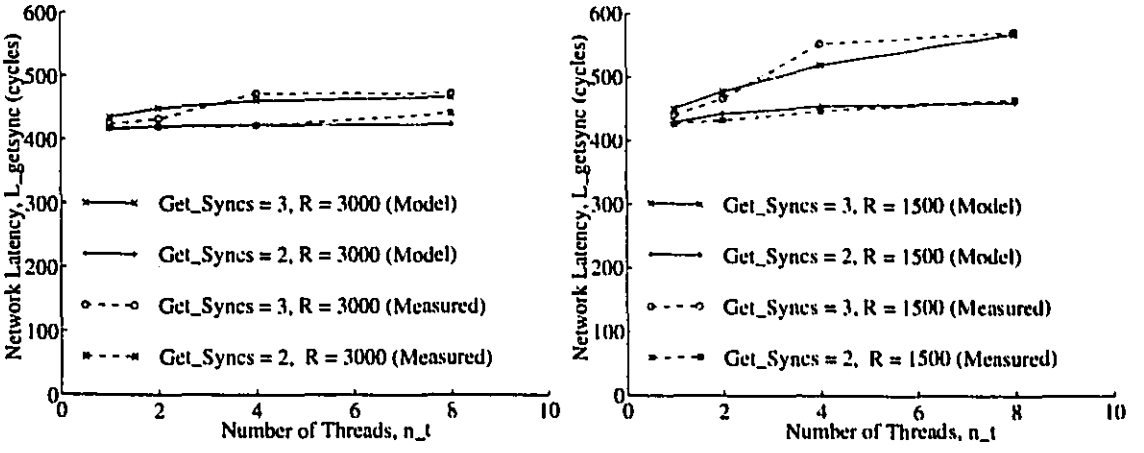


Figure 8.4: Effect of the number of remote accesses per thread on $L_{get-sync}$.

many functional units on the path of a remote access. Figure 8.5 shows how the delay at the remote SU varies with n_t and R. Continuous lines are model predictions for R = 3000 cycles, and dotted lines are model predictions for R = 1500 cycles. The delay at the SU increases with n_t. A more significant increase occurs with the number of GET_SYNC operations per thread; for example, when the number of GET_SYNCs is 3, each GET_SYNC requires a service of 20 cycles at the SU and the overall delay is 60 cycles. For threads with low R (=1500 cycles), the queueing delay at an SU increases rapidly. So, a high $L_{get-sync}$ occurs as noted in Figure 8.4. Finally, we note that at high thread runlength, the delay at SU does not change significantly with the number of threads.

Figure 8.6 shows how the local read/write accesses rw affect $L_{get-sync}$. We consider rw = 1 and 4. Local read/write accesses increase $L_{get-sync}$ by less than 2% to 3% in all cases. There are two reasons as follows: *First*, unlike remote accesses, service times for local accesses are very small. For example, a local memory access requires 15 cycles at the memory. In contrast, a remote memory access issued by an EU initiates 4 accesses to its local memory, two to initiate the remote memory access, and two to complete the remote memory access. *Second*, each split-phase multithreading operation causes multiple bus accesses (to memory and network interfaces). So, a small number of read/write accesses per thread does not significantly increase the contentions on the node bus. We report the system measurements only. Model predictions for $L_{get-sync}$ also change within 2% to 3% (not shown in Figure 8.6).

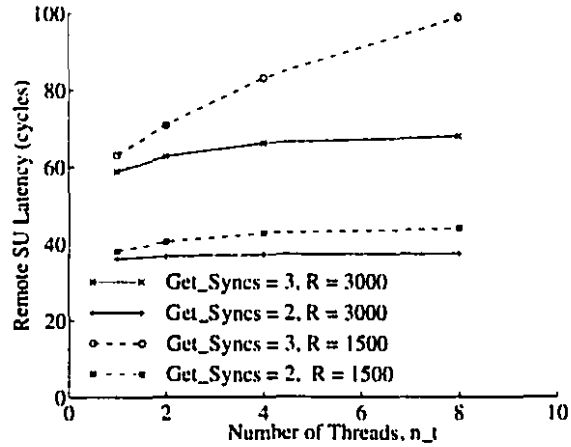


Figure 8.5: Effect of program workload parameters on the delay at the SU.

We show the effect of local read/write accesses on U_p in Figure 8.7. Similar to the changes in $L_{get-sync}$ values, U_p values decrease by less than 5% due to increase in local accesses.

In summary, we note how sensitive the performance of real multithreaded machines is to the number of multithreading operations and thread runlengths.

- The number of GET_SYNC operations per thread is significant to determine the contentions in the system. Even with as small as 3 GET_SYNC operations per thread, $L_{get-sync}$ for individual accesses is high on a real system like the EARTH.
- Delays at functional units like the SU in a processing node dominate the latency costs, in the current EARTH implementation. As a result $L_{get-sync}$ increases with the number of GET_SYNCs per thread. In spite of these long delays at functional units, the processor performance increases under multithreaded execution compared to a single threaded execution.
- A program workload achieves high processor utilization (> 80%) on the EARTH system for thread runlengths above 3000 cycles (60μs) and the number of GET_SYNCs is less than 3. When the number of GET_SYNCs per thread increases, the contentions increase and the U_p values decrease.
- The number of local read/write accesses do not significantly affect the performance

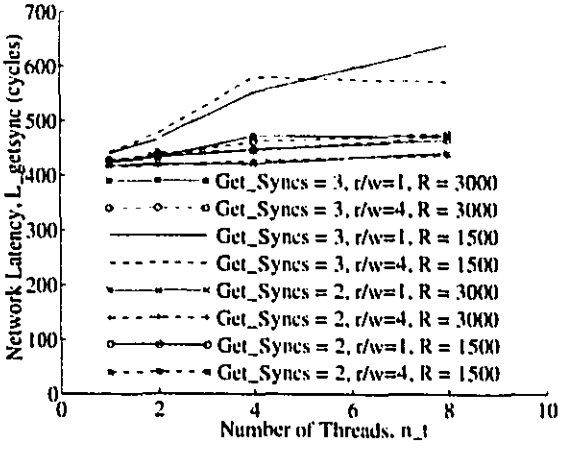


Figure 8.6: Effect of remote and local accesses on $L_{get-sync}$.

of the EARTH system, as compared to the number of remote accesses. Changes in the number of read/write access to local memory change $L_{get-sync}$ and U_p values by less than 5%.

8.4 Architectural Optimizations

The delays at a processing node reduce the performance gains achieved due to multithreading as shown by Section 8.3.2. Now, we discuss how different architectural implementations of the EARTH system can yield higher performance gains. We consider two systems. First, the system has the network delays similar to those observed in a *Networks-of-Workstations* type system. Second, the processing node overheads in the EARTH system are reduced. We use model predictions for comparing the performances of all systems. Predictions for the current EARTH system are validated earlier in this chapter.

One of the recent trends in large scale computing is to use *Networks of Workstations* (*NOW*) to achieve high throughputs [11, 71]. Some of the characteristics of these systems in comparison to multiprocessor systems are as follows. The processing nodes in a *NOW* system are similar to those used in a multiprocessor system like the EARTH. However, the interconnection among these nodes in a *NOW* system is slower than the network in a multiprocessor system. A *NOW* system uses standard interconnection networks, like ATM or Ethernet. Also the distances between workstations can be large. So, communication la-

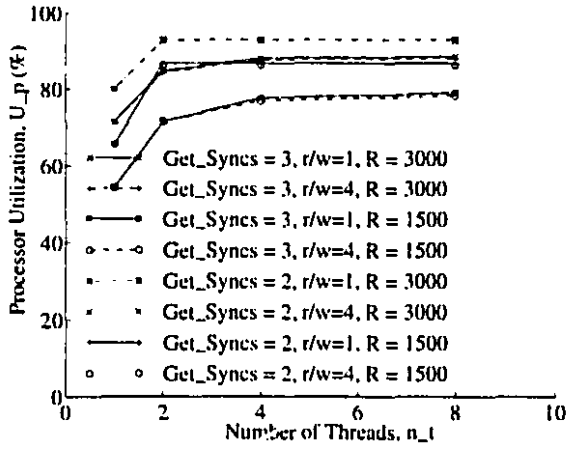


Figure 8.7: Effect of remote and local accesses on U_p.

tencies for messages are higher than those in a multiprocessor system, which use customized interconnection networks.

For our discussions of a *NOW* system, we consider that the processing nodes of the EARTH system are connected using an interconnection network slower than the one in the EARTH system. We use the best values for a round-trip latency reported by Pakin et al [71]. In particular, the round trip latency is $50\mu s$ (and one-way is $25\mu s$). To achieve this value, we need to set the crossbar delay $xout_{serv}$ to be 1100 cycles. The rest architectural parameters have the same values as on the EARTH system (see Table 8.1).

Figure 8.8 shows how U_p values of the *NOW* system compare with U_p for the EARTH system, when the number of threads are increased. Continuous lines show the performance of the current implementation of the EARTH system on MANNA platform. Dotted lines show how the performance of the *NOW* system changes with thread runlengths, when the number of `GET_SYNC` operations per thread and thread runlengths vary. As expected, the single-threaded performance of the *NOW* system is worse than the EARTH system, because the delays on the network are much longer. However, with multithreading the performance of the *NOW* system improves rapidly. Let us consider that R is 1500 cycles. When the number of `GET_SYNCs` is 3, with increasing n_t, U_p values saturate close to double the value at $n_t = 1$. With `GET_SYNCs` = 2, the U_p for $n_t = 8$ is almost 2.5 times the U_p at $n_t = 1$. When R is 1500 cycles, U_p saturates at 24% and 46% with `GET_SYNCs` = 3 and 2 respectively, because the network latencies ($L_{get-sync}$) are very high. For R= 3000 and 6000 cycles, the

saturation of U_p occurs at 80% and above. Further, at R= 3000 cycles, U_p rises to nearly 3 times at n_t=16 (not shown in Figure 8.8). Similar performance gains are observed even when the number of remote accesses per thread.

In summary, with current service times of subsystems in the EARTH processing node, a 8-node *NOW* system can achieve as much as 3 times the performance of a single threaded program execution. Interestingly, thread granularities, which yield high processor utilization, are almost same as those we noted for the current EARTH implementation, i.e. thread runlengths of 3000 cycles and above.

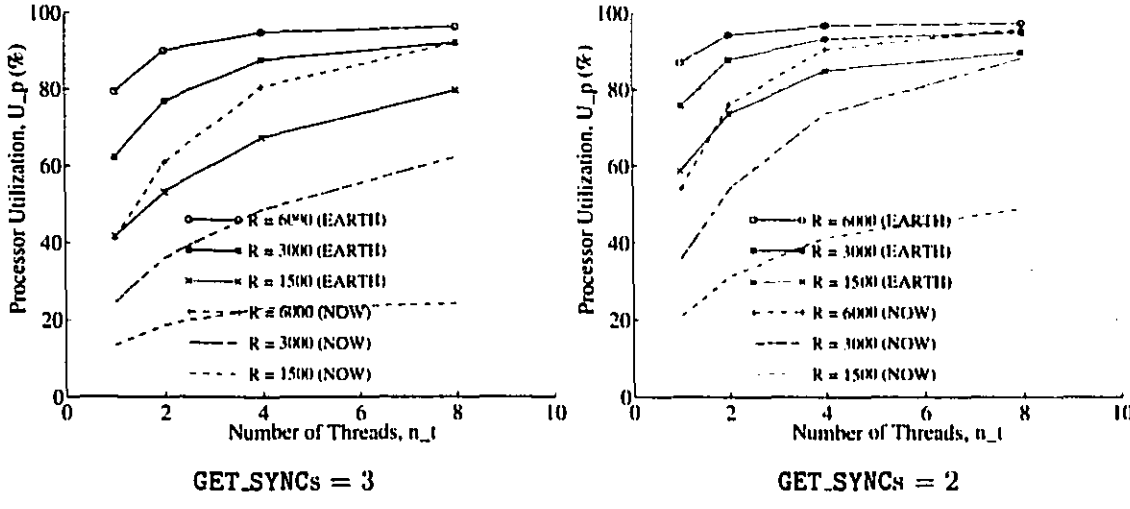


Figure 8.8: U_p for a *NOW* system.

Let us consider another important trade-off in the EARTH processing node design, service times at subsystems like the SU and the link interfaces. We compare the performance of systems with the current EARTH node implementation (whose service times are mentioned in Table 8.1) and with a hypothetical *fast* EARTH node. The service times at the *fast* EARTH node are as follows: SU_{serv} is 10 cycles; $lnkin_{serv}$ is 8 cycles; and $lnkout_{serv}$ is 4 cycles. The model predictions of U_p for the current EARTH implementation and the *fast* system are shown in Figure 8.9. Continuous lines with the label **EARTH** are the model predictions for the current EARTH system. Dotted lines with the label **Fast** are the model predictions for the *fast* EARTH system.

For single threaded program execution, we observe that a fast system yields nearly 10% higher U_p than the current EARTH system when R is 1500 cycles (Figure 8.9). At higher

thread runlengths, there is not much scope of performance improvement due to high U_p values. However, the performance improvement decreases with an increase in n_t, because latencies are better tolerated with multithreading.

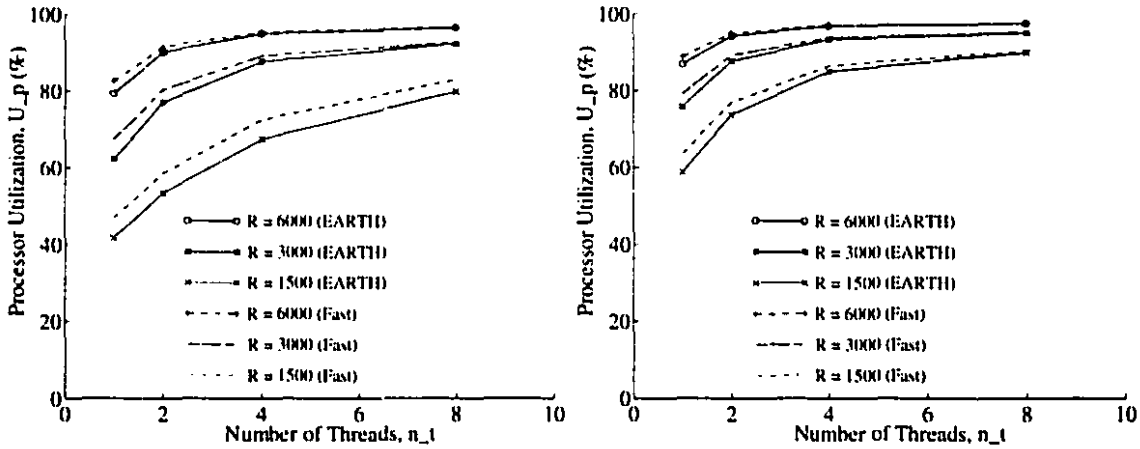


Figure 8.9: Effect of fast subsystems on U_p.

Figure 8.10 shows network latencies $L_{get-sync}$ for the two systems, the *fast* EARTH system and the current EARTH system. $L_{get-sync}$ values for the fast system are at least 100 cycles lower than $L_{get-sync}$ for the current EARTH system. However, $L_{get-sync}$ values for the *fast* EARTH system follow the same trend with model parameters as the $L_{get-sync}$ values for the current EARTH system. That is, $L_{get-sync}$ increases rapidly with n_t when R is small and the number of GET_SYNCs per thread is 3 or more.

In summary, we note that lower latencies for the *fast* EARTH system yields up to 10% higher U_p values than the current EARTH system, when thread granularities are low. This result conforms with the common notion that a faster communication enables an efficient execution of finer grain parallel program. However, with multithreading the performance gap narrows down to less than 5% between the *fast* EARTH system and the current EARTH system.

The next section explores an advantage of multithreading on a distributed shared memory multiprocessor systems. The ability of multithreading to tolerate latency enables us to achieve better performance even in the absence of a data locality.

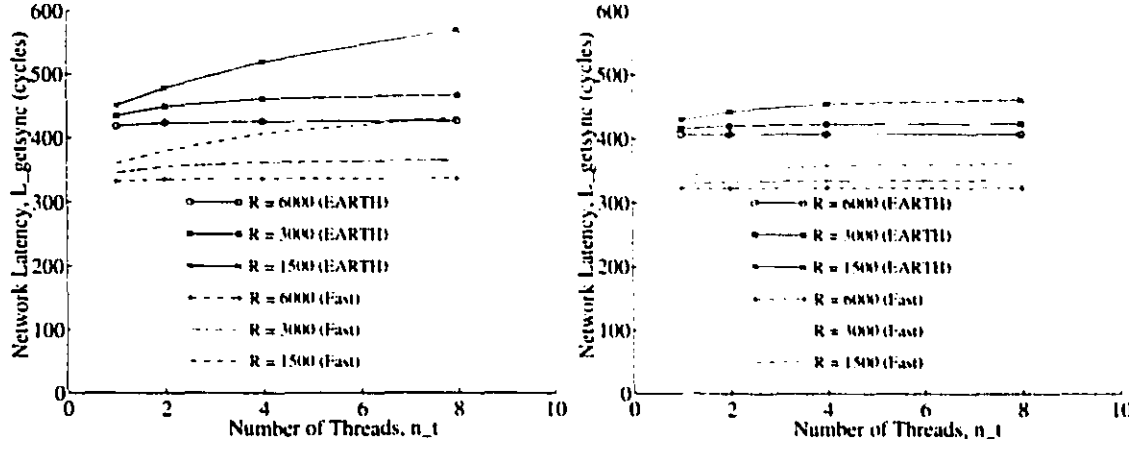


Figure 8.10: Effect of fast subsystems on $L_{get-sync}$.

8.5 Data Locality Sensitivity

In this section, we investigate the robustness of the performance of the EARTH system to changes in data distributions. First, we define metrics to evaluate the sensitivity of performance to changes in data distributions. Second, we describe our program workload to study the data locality sensitivity. Third, we show results on the data locality sensitivity of single-threaded and multithreaded program workloads.

8.5.1 Metrics for Data Locality Sensitivity

Given a multiprocessor system like EARTH, the data locality sensitivity of a program is the variation in the performance of the system due to changes in data distributions. We define two metrics to quantify the data locality sensitivity of distributed memory multiprocessor systems. These metrics are, the *multiprocessor locality sensitivity index (MLSI)* and the *locality sensitivity index (LSI).*

Intuitively, the worse the locality of the data access pattern, the worse the performance of single-threaded multiprocessor systems. A lower locality in a data distribution increases the latency for a remote memory access (see Section 5.7). A processor waits for the long latency access to complete before executing further. To get a reasonable performance, a

compiler or a programmer needs to spend efforts to minimize inter-thread communication and maximize the reuse of the data in local memory.

Changes in data distributions change the network latency experienced by an access. A deleterious effect of a remote memory access on performance can be reduced, if this remote memory access latency is overlapped with an execution on other threads, i.e. the multithreading support. Thus, the multithreading should reduce the needs of compiler writers and programmers to carefully craft data distributions to achieve high performance. This is especially beneficial for irregular and communication intensive applications, where an optimal data distribution may not be easy to find.

The performance of a multithreaded system also depends on a matching of program workload to the underlying machine architecture. We focus on the following program workload parameters: (a) the number of threads n_t at each processing node; (b) the thread runlength R; (c) the number of long latency accesses per thread; (d) the probability p_{remote} of sending these accesses to remote memory; and (e) the number of processing nodes P. On the EARTH system, we use the program execution time to compare the performance of various program workloads and data distributions.

Next, we define metrics to quantify data-locality sensitivity.

Definition 1 *Given a program workload, let D_P be a well-tuned data distribution on P processoring nodes (i.e. with an optimal data locality), and D_P^* be the other data distribution under investigation, e.g., a randomly distributed data. Let $T_{P,n_t}(D_P)$ and $T_{P,n_t}(D_P^*)$ be execution times of a program running on P nodes of a multiprocessor system, when data distributions are D_P and D_P^*, respectively. The Multiprocessor Locality Sensitivity Index (MLSI) is defined as:*

$$MLSI_{P,n_t}(D_P, D_P^*) = \frac{T_{P,n_t}(D_P)}{T_{P,n_t}(D_P^*)} \tag{8.1}$$

where n_t denotes the average number of threads on each node. A data distribution is optimal, when no remote memory access is required for any data accessed by threads at a processor, except the essential *remote data accesses.* Essential *remote data accesses are the accesses for data that must be shared among threads at different processors.*

The MLSI measures how is the performance of a system on a program with a particular data locality. The MLSI compares the performance for a program workload on an imperfect

data locality, with the performance on a perfect data locality, i.e., optimal distribution. The MLSI usually ranges between 0 and 1. Note that MLSI may be greater than 1, if D_P^* leads to lower program execution time than D_P. The closer the MLSI value to 1, the more robust the performance with respect to variations in the data locality. (For some programs, it is not difficult to obtain an obvious, optimal distribution.) We can also use the MLSI to compare different data distributions, for example, how is the performance on a block distributed partitioning of matrices in an application with a block cyclic distributed one.

We note the similarity of the MLSI with the Tolerance Index discussed in Chapter 6. There are following two major differences:

1. The MLSI measures the performance variation of a system on a program with respect to different data distributions. There is a possibility that the optimal distribution D_P may not necessarily be the one with all data residing in local memory module. In contrast, given a program workload and a data distribution, the tolerance index (for network latency) compares the performance of a system with that of an *ideal* system, i.e., a system with no delays on the network.

2. The MLSI helps to assess the performance variations of a distributed memory multiprocessor system with respect to variations in the data locality. In contrast, the tolerance index helps to assess the impact of the latency of a subsystem on the performance of the system on given program workload.

Now, we define another metric Locality Sensitivity Index (LSI), which is a special case of the MLSI. The LSI deals with the program execution on a single processing node, when the data is distributed on P processing nodes. Thus, the LSI removes the impact of contentions at the network on the performance comparison. In case of the EARTH system, the LSI also reduces the effect of waits at remote processing nodes to service long latency accesses.

Definition 2 *Given a program workload, let D_P be a well-tuned data distribution on P processoring nodes (i.e. with an optimal data locality), and D_P^* be the other data distribution under investigation, e.g., a randomly distributed data. Let $T_{1,n_i}(D_P)$ and $T_{1,n_i}(D_P^*)$ be execution times of a program running on 1 node of a multiprocessor system, when data distributions are D_P and D_P^*, respectively. The Locality Sensitivity Index (MLSI) is defined*

as:

$$LSI_{n_t}(D_P, D_P^*) = \frac{T_{1,n_t}(D_P)}{T_{1,n_t}(D_P^*)} \tag{8.2}$$

where n_t denotes the average number of threads on each node. A data distribution D_P is optimal, when no remote memory access is required for any data accessed by threads at a processor.

The LSI is a positive real number between 0 and 1. The closer the LSI value to 1, the more robust the performance of a system on a program with respect to variations in data locality. The LSI reflects how good is the performance of a node in a multiprocessor system, when the data is distributed on multiple nodes. In other words, the LSI indicates the ability of a node in a multiprocessor system to tolerate the remote access latency. Again, we notice a similarity with the tolerance index that the optimal data distribution D_P for the LSI ensures the absence of remote accesses.

In conventional parallel computing, the performance of a multiprocessor system on a single-threaded program workload changes significantly due to the data locality. So, compilers and programmers must spend significant efforts to improve data locality through smart data partitioning strategies and changes in the control structure of program workload. Our intuition is that the performance of multithreaded computation is less sensitive to changes in data distributions. Accordingly, we study how robust is the performance of a multithreaded system with respect to variations in data locality.

Next, we describe program workloads and their data distributions on the EARTH system. We use these program workloads to investigate the data locality sensitivity of the EARTH system in Section 8.5.3.

8.5.2 Program Workloads and Data Distributions

We introduced the performance metrics, MLSI and LSI, to quantify the data locality sensitivity. In this section, we show how to compute the MLSI and LSI for a specific application, and also compare performance of the single-threaded computation with multithreaded computation. The empirical measurements are made under a synthetic workload with different data distributions. We also outline program workloads used to study the data locality sensitivity of the EARTH system.

MLSI and LSI Computation

Now, we show how to compute the MLSI and LSI for a synthetic benchmark. Later, we extend this computation to Matrix Multiplication.

Synthetic Benchmark

The synthetic benchmark is similar to the program workload shown in Figure 2.2 (Section 2.2). We vary the number of threads n_t forked on each processing node and their thread runlengths R. All input data for a thread is fetched from remote memory module before the computation begins. We use BLK_MOV operations to perform these long message transfers between remote processing nodes. The size of network messages changes depending on the granularity of threads. We also vary data distributions, i.e., change p_{remote}.

The synthetic benchmark SB provides us a flexibility to adjust the communication pattern and program execution behavior. We measure measure MLSI and LSI for different data distribution, and for variations in input parameters such as n_t and p_{remote}. The threaded function of SB mainly consists of two parts: (a) a communication-thread which reads the data from local/remote memory; and (b) a computation-thread which serves to compute the result and write back. We propose the following four computation-communication patterns to measure the MLSI and LSI.

- *Single Thread on 1 node (ST-1)*: The processing node 0 executes one thread, i.e., $n_t = 1$. The input data for the computation is distributed on P processing nodes. This thread issues two remote read operations (using BLK_MOVs) for the input data. On return, the computation part of this thread is triggered. This process is repeated till the end of the computation.

- *Multiple Threads on 1 node (MT-1)*: The processing node 0 executes n_t thread. The input data for the computation is distributed on P processing nodes. Each thread issues two remote read operations (using BLK_MOVs) for the input data. On return, the computation part of the waiting thread is triggered. This process is repeated for each thread till the end of the computation.

- *Single Thread on P nodes (ST-P)*: Each processing node i (where $i = 0, 1, ..., P-1$) executes one thread, i.e., $n_t = 1$. The input data for the computation is distributed on

P processing nodes. This thread issues two remote read operations (using BLK_MOVs) for the input data. On return, the computation part of this thread is triggered. This process is repeated till the end of the computation.

- *Multiple Threads on P nodes (MT-P)*: Each processing node i (where $i = 0, 1, ..., P-1$) executes n_i thread. The input data for the computation is distributed on P processing nodes. Each thread issues two remote read operations (using BLK_MOVs) for the input data. On return, the computation part of the waiting thread is triggered. This process is repeated for each thread till the end of the computation.

Figures 8.11 (a) and (b) show the execution patterns for ST-1 and MT-1. Let us compute the performance benefit of multithreading using a simplistic back-of-the-envelope analysis based on simple measurements from the EARTH system. The timing details are presented to illustrate the concept.

In Figure 8.11 (a), let the BLK_MOV operation take $120\mu s$ ($=L_{blkmov}$) for ST-1. The EU at processing node 0 takes $1\mu s$ ($=I$) to issue a BLK_MOV. Let us assume that thread runlength R is $300\mu s$. The context switch time C of $2.5\mu s$ is incurred because the first part of thread issued remote accesses, and the second part gets triggered at the completion of these accesses. Then the overall execution time T_a is $(I + C + L_{blkmov} + R) = 425.5\mu s$. All of these values are measured from the EARTH system.

Now, for MT-1 in Figure 8.11 (b), we split the computation into 3 threads. Since the message size requirement for each thread is reduced to one-third of that in Figure 8.11(a), we assume that the latency for each BLK_MOV reduces to $40\mu s$ ($= L_{blkmov}$). For the Thread-1 (the left-most one), we have $I = 1\mu s$, $C = 2.5\mu s$ and $R = 100\mu s$. The thread completes execution after a duration of $(I + C + L_{blkmov} + R) = 143.5\mu s$. The Thread-2 (in the middle) receives its BLK_MOV operation by $136.5\mu s$. Though we expect that this BLK_MOV by $I + C + L_{blkmov} + L_{blkmov} = 83.5\mu s$, a contention between the EU and the SU at processing node 0 delays the completion of the second BLK_MOV operation. The execution on Thread-2 follows. Thus, Thread-2 completes after a duration of $(I + C + L_{blkmov} + R + C + R) = 246.0\mu s$. For Thread-3 the same logic applies regarding the contention between the EU and the SU. The corresponding BLK_MOV operation completes by $235.5\mu s$. So, the overall execution time of the program is $T_b = I + C + L_1 + R + C + R + C + R = 348.5\mu s$. y A comparison of execution times in two cases T_a and T_b shows the benefit

due to multithreading. The performance improvement is $(T_a - T_b)/T_a = 0.181$ i.e., 18.1%. We note that in Figure 8.11(b), the responses to latter two `BLK_MOV`s are received before the processing on `Thread-1` and `Thread-2` complete. If the distribution of data is worse, leading to a higher latency, the overlap of computation and communication ensures a smaller variation in the performance.

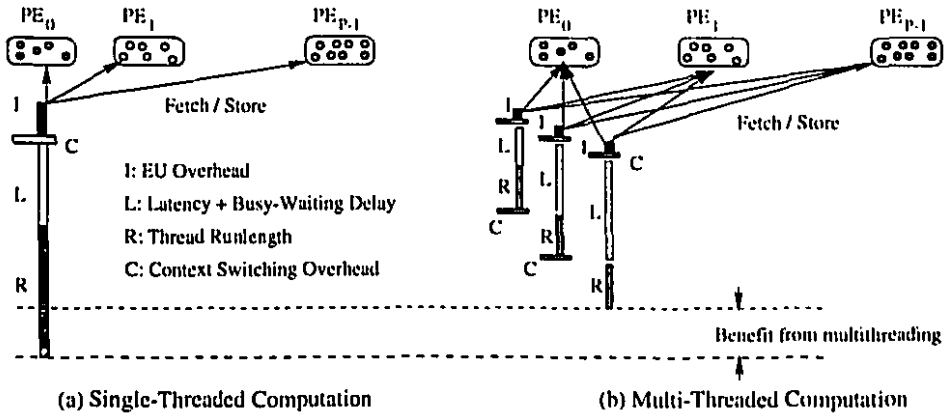


Figure 8.11: Examples of ST-1 vs. MT-1 Execution Patterns

Executions for ST-P and MT-P are very similar to those shown in Figure 8.11. The execution of ST-P differs from that of ST-1 as follows. While the ST-1 allows execution of single thread only on processing node 0, the ST-P allows each processing node to execute one thread. Similarly, MT-P allows a number of threads $n_t(> 1)$ on each processing node in contrast to MT-1, which allows n_t threads only on the processing node 0. Thus, ST-1 and MT-1 incur no contention at remote nodes for remote memory accesses (but contentions occur at the network and the processing node 0 among multiple outstanding accesses from different threads at processing node 0). On the other hand, under ST-P and MT-P, remote access from a processing node incur contentions at the network and at each processing node with remote accesses from other processing nodes. The ST-1 pattern leads to a shorter delay for busy-waiting on each remote memory access compared to the ST-P execution pattern, because the ST-1 execution occurs only on the node 0 and there are no threads to synchronize (on node 0 or other nodes). The same argument applies to the MT-1 and MT-P execution patterns.

Now, we describe the Matrix Multiplication, which we will use to study the data locality

sensitivity of the EARTH system. We have implemented a 1-D Systolic Matrix Multiplication (SMM) algorithm to compute $C = A \times B$. A, B and C are $n \times n$ matrices, and $N = n^2$. Let us assume that P processing nodes exchange data as they would on a ring. Let each processing node have w consecutive columns, where $w = \frac{n}{P}$. Thus, $A = [A_0\ A_1\ ...\ A_{P-1}]$, where A_i block is located on the i-th processing node at the start of computation. A similar distribution exists for $B = [B_0\ B_1\ ...\ B_{P-1}]$. As the computation proceeds, C_k accumulates on the processing node k, such that $C = [C_0\ C_1\ ...\ C_{P-1}]$. The progress of computation is as follows: *First*, each node k computes $A_k \times B_k$ and stores as a partial sum for C_k. *Second*, A_k's are cyclically shifted, say to nodes $(k+1)$. *Third*, the partial computation of C_k repeats at the node k. The second and third steps continue, till A_k visit all processing nodes (a total of $(P-1)$ shifts). Thus, C_k accumulates at the node k. That is, $C_k = \sum_{i=0}^{P-1} A_i B_{ik}$, where B_{ik} contains rows $i * w$ to $(i+1) * w$ of B_k.

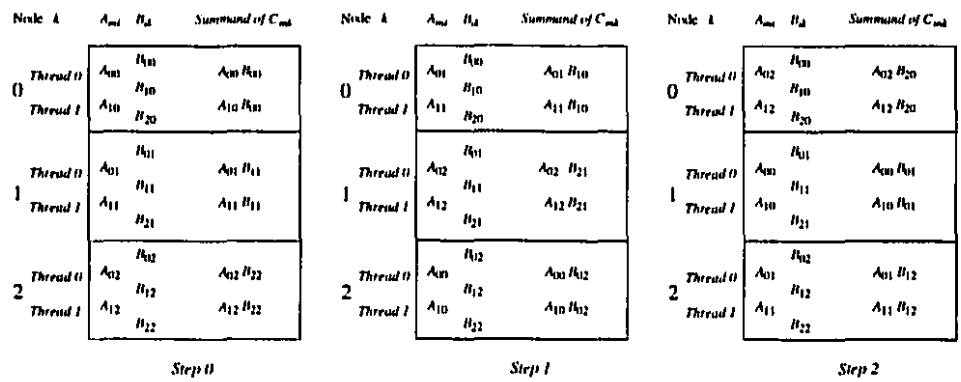


Figure 8.12: Execution Pattern of 1-D Systolic Matrix Multiply (SMM)

Let us consider the data movement in single-threaded and multithreaded execution of the SMM. For a single-threaded execution, each cyclic shift of A_k is followed by a partial computation of C_k on the node k. Total number of shifts are $(P-1)$. On each shift, a node receives $\frac{n^2}{P}$ data elements and performs $\frac{n^3}{P^2}$ multiplications on matrices of sizes $n \times \frac{n}{P}$ and $\frac{n}{P} \times n$. The computation and communication phases of the program are distinctly visible. For a multithreaded program execution, each processor executes on n_t threads after each cyclic shift of A_{mk}, m= 1,2,...,n_t. Rows numbered between $m * w$ and $(m+1) * w$ of A_k and C_k are referred as A_{mk} and C_{mk}. Each thread requires $\frac{n^2}{n_t P}$ data elements on each shift, and performs $\frac{n^3}{n_t P^2}$ multiplications (and their additions) on matrices of sizes $\frac{n}{n_t} \times \frac{n}{P}$

and $\frac{n}{p} \times \frac{n}{p}$. Thus, after the arrival of the first block A_{1k} of $\frac{n^2}{n_t p}$ elements, the computation on threads is overlapped with the arrival of remaining $(n_t - 1)\frac{n^2}{n_t p}$ data elements. With multithreading, the computation and communication phases are overlapped thereafter.

8.5.3 Data Locality Sensitivity of the EARTH System

In this section, we will investigate how sensitive is the performance of the EARTH system to changes in data locality. First, we study the data locality sensitivity when the EARTH system executes the synthetic benchmark discussed in Section 8.5.2. Second, we study the data locality sensitivity for the Matrix Multiplication program discussed in Section 8.5.2.

Now, we explore the locality sensitivity varies with various workload parameters. Synthetic benchmark in Section 8.5.2 allows us to vary the number of threads, their runlengths, and their data access patterns. We consider the effect of workload parameters on the Multiprocessor Sensitivity Index, MLSI, shown in Figure 8.13. The synthetic benchmark is executed on 20 processing nodes. The block size for remote data accesses (`BLK_MOVs`) is such that the program performance is the least when each processing node executes one thread. These block sizes for each `BLK_MOV` are 120 and 480 floating point numbers. The number of threads n_t on each processing node varies from 1 to 20. The fraction, p_{remote}, of long latency operations sent to remote memory varies from 0 to 0.95, as shown on the axis titled "Data Distribution" with a block size of 120 floating point numbers. Note that the remote data is distributed on remaining nodes as follows: for $p_{remote} = 0.5$, one node contains rest of the data; for $p_{remote} = 0.66$, two nodes contain rest of the data; for $p_{remote} = 0.9$, nine nodes contain rest data; and so on. For a particular value of n_t (say 2), the MLSI is the relative decrease in program execution time due to a data locality ($p_{remote} > 0$) with respect to the program execution time when the data is local i.e. $p_{remote} = 0$. We observe that:

- For the single-threaded execution ($n_t = 1$), the MLSI decreases by 18.8% when all of the data is distributed over remote memory modules. When half the data is remotely located, the decrease in MLSI is nearly 14.7%.
- For a multithreaded execution, the decrease in the MLSI is less than 5% when p_{remote} is greater than 0.5. The closeness of MLSI values to 1 for $n_t > 1$ shows that the locality has very little impact on the performance of multithreaded program workload.

Program execution times for these MLSI values shown in Figure 8.13 indicate that $n_t = 2$ and 4 provide the best performance. At high n_t (8 to 20), the program execution time is up to 8.5% higher than that for $n_t = 2$, because an extra time is required to fork the threads at the beginning of computation and to synchronize the threads at the end of computation.

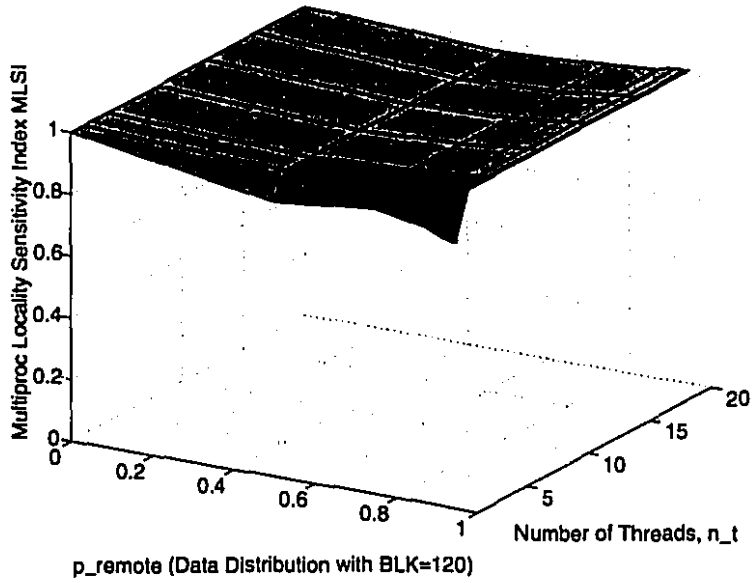


Figure 8.13: MLSI values for Synthetic Workload at BLK=120.

Figure 8.14 shows how the processor performance varies when there is no contention from executions on other processors. The synthetic benchmark program executes on the processing node 0, and the data is distributed on 20 nodes in the system. In Figure 8.14, a long latency BLK_MOV is sent to a remote memory with the probability p_{remote}. As mentioned above, with $p_{remote} = 0.5$, node 1 contains rest data; with $p_{remote} = 0.75$, nodes 1, 2 and 3 contain rest data; and so on. p_{remote} values are 0, 0.5, 0.75, 0.875, 0.917, 0.925, and 0.95, corresponding to the number of nodes 1, 2, 4, 8, 12, 16, and 20 on which the data is distributed. Each BLK_MOV transfers 480 elements, one of the two best sizes for single-threaded performance on a 20-node EARTH system. The number of threads on the node 0 vary from 1 to 20. From Figure 8.14, we note the following:

- A distribution of data to remote memory modules decreases LSI values by 17.4 to 20.3% when $n_t > 1$. However, an increase in p_{remote} from 0.5 to 0.95 marginally increases the LSI value. The reason is that at higher p_{remote} the data is spread over

a larger number of nodes, so bottlenecks at remote nodes to service remote memory accesses reduce.

- For $p_{remote} \geq 0.5$, with multithreading the LSI values increase by 19.% over the LSI values at n_t=1.

Similar to the observation for the MLSI values, we note that $n_t = 2$ provides the best program execution time as well as the highest LSI values. The reason is that when a remote access is in progress for one thread, the computation on another thread hides the latency of remote access. When there are more than 2 threads (say 10 or 20), a large time is also spent in forking these threads before the computation begins and synchronizing them at the end of computation.

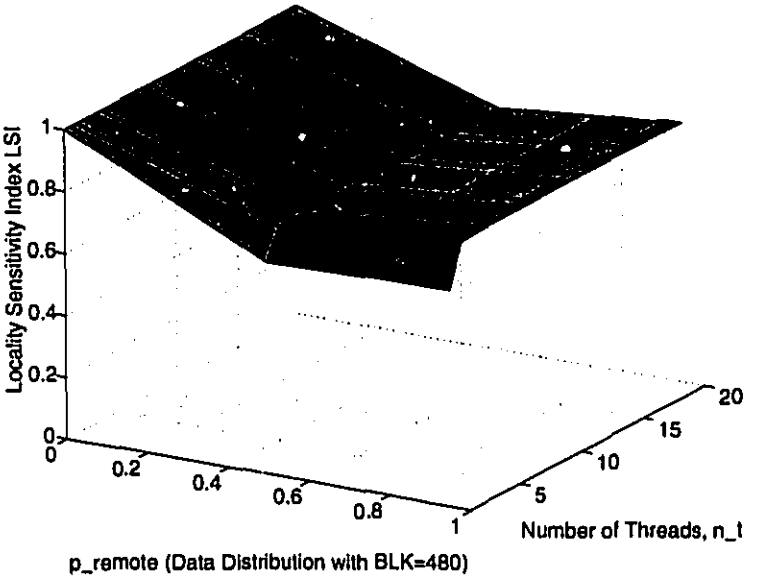


Figure 8.14: LSI values for Synthetic Workload at BLK=480.

Next, we consider the matrix multiplication program (SMM) discussed in Section 8.5.2. The MLSI values for the SMM are plotted in Figure 8.15 with respect to the number of threads n_t per processing node and different data distributions, i.e., different p_{remote} values. n_t varies from 1 to 20. p_{remote} values are 0, 0.5, 0.75, 0.875, 0.917, 0.925 and 0.95. Given a p_{remote}, the smallest MLSI value occurs at $n_t = 1$. With increasing $n_t \geq 4$, the MLSI values are almost constant. The program execution times (not shown in Figure 8.15) are the least

for $n_t = 4$ and 8. We also note that for $p_{remote} \leq 0.917$ the MLSI values are almost close to 1 when $n_t \geq 4$. Comparing the MLSI values for a single threaded execution ($n_t = 1$) to those for a multithreaded execution, we observe the following:

- The MLSI decreases rapidly with increasing p_{remote}, especially when $n_t = 1$.
- At large p_{remote} (say 0.95), the MLSI value increases by as much as 11.8% with multithreading. (The program execution time decreases by nearly 14.5%. However, the program execution time decreases for $n_t > 1$ even when $p_{remote} = 0$, so a lower improvement in the MLSI value is observed.)

Overall, our experiments show that the decrease in MLSI (and LSI) for a multithreaded program execution is less than the decrease in MLSI (and LSI) for a single threaded execution. So, the multithreaded program execution is more robust to the changes in data distributions. Thus, we believe that an inherent ability of multithreading technique to tolerate long latency helps to maintain a high performance. A user of multithreaded system needs to make only a small effort on data distribution even when the remote access pattern is communication-intensive and irregular or changes dynamically in a program execution.

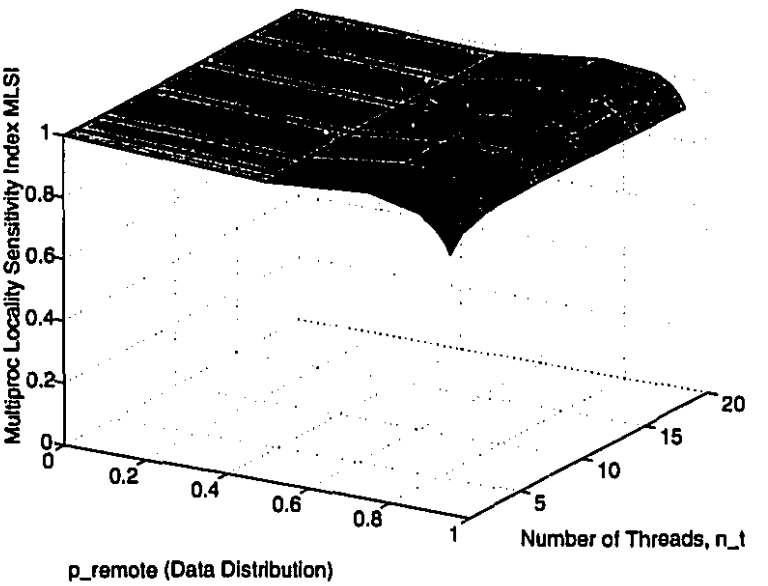


Figure 8.15: MLSI values for Matrix Multiplication.

8.6 Related Work

In this chapter, we have focussed on the issues related to the performance optimization of multithreaded multiprocessor systems. We characterized the performance of the EARTH system. Then we addressed the sensitivity of performance to the changes in data locality.

First, we characterized the performance of the EARTH system with workload parameters. Arpaci *et al* [13] report a characterization of latencies for various operations on CM-5 using Split-C. Boyd and Davidson [19] report an evaluation of a multiprocessor system using synthetic benchmarks. One of the main objectives of these studies is to provide information on how much penalty user-level language primitives incur. These primitives are various types of local and remote accesses. Compilers and programmers can choose the best possible hardware mechanism using these primitives to achieve faster communication. However, they do not consider the impact of overlap of computation and communication on performance measures. We show in this chapter that the overlap has significant effect on the processor performance. Two important aspects of the overlap are as follows. First, the overlap improves the processor performance compared to a single-threaded program execution. Second, the number of outstanding requests per processor increase with the number of threads, so strategies for performance related optimizations change.

On the other hand, Woo et al [102] study SPLASH-2 benchmark suite, specifically the aspects like the application parallelism, computation to communication ratio, and locality. They use above aspects to characterize a program execution on a multiprocessor system. However, they do not consider how the architectural mechanisms affect the performance of a multiprocessor system on a program workload. Our results show that program workload parameters as well as architectural parameters should be considered to analyze and optimize the performance of a program workload on a multithreaded multiprocessor system like the EARTH.

Second, we investigated the data locality sensitivity of performance of the EARTH system. Some of the related work are by Johnson [50], Felten [36] and Sohn [89]. Johnson proposed a model in [50] to predict the performance of a multithreaded system. His model characterizes application behavior with parameters that capture the computation granularity, the sensitivity to communication latency and the degree of locality in the access pattern. He shows that exploiting communication locality provides gains on overall performance. In

other words, a high locality in remote accesses helps to achieve a good performance on large-scale multiprocessor systems, and the performance is sensitive to the communication locality. The results of this chapter show that with multithreading, the performance is more robust than single threaded systems to the changes in locality in remote accesses.

Felten and McNamee [36] argued that computation and communication overlap can be easily achieved by executing multiple threads on each processor. They also argued that this approach is practical on distributed-memory architectures without any special hardware support. They presented timing data for the PDE solver [36]. Their results conform with the data-locality analysis in this chapter.

Sohn et al [89] investigated the effects of multithreading on data distribution and workload distribution. On 80-processor EM-4 distributed memory multiprocessor, they investigate three types of data distribution, namely, row-wise cyclic, k-way partial-row cyclic, and blocked distribution. Their experimental results indicated that multithreading can offset the loss due to a mismatch of distribution to workload distribution. This work is perhaps the closest to our results on data locality sensitivity. We show that the ability of multithreaded program execution to tolerate long latency provides a robustness of performance to the changes in the data distribution of a program workload.

8.7 Summary

In this chapter, we investigated various aspects on the optimization of the performance of the EARTH system.

First, we showed how various performance measures are affected when parameters of a program workload are varied.

Second, we investigated the trade-offs for thread runlengths, number of remote accesses per thread and number of local accesses per thread. We noted that the larger the number of remote accesses per thread, the lower the processor utilization. The performance can be improved with higher thread runlength and not by higher number of threads. Changes in the local read/write accesses do not affect the processor utilization and latencies significantly.

Third, we defined measures to quantify the sensitivity of data locality to the performance. We applied these measures, the multiprocessor locality sensitivity index (MLSI)

and the locality sensitivity index (LSI), to study the EARTH system. We noted that even with current implementations of the EARTH system, the MLSI (and LSI) values for multithreaded program workload are up to 20% higher than those for single threaded execution. That is, the multithreaded program execution significantly increases the robustness of performance to the changes in data distributions. The reason is that a multithreaded execution is expected to tolerate long latencies, which can be a result of changes in data distributions. The implication of our result is that a programmer/compiler need not have to achieve the optimal data distribution (which may not even exist) on a multiprocessor system and yet reach near the performance levels of an optimal distribution.

Chapter 9

Conclusions

This chapter summarizes main results of this thesis and outlines directions for future work.

9.1 Summary

This thesis showed that the performance analysis of multithreaded systems differs significantly from that of traditional single-threaded systems and systems with a multitasking operating system. A multithreaded processor is capable issuing multiple outstanding requests. A user of multithreaded systems needs to assess the following tradeoff: a performance improvement due to an increased overlap of computation and communication; and a performance loss due to contentions and increased latencies at subsystems. To satisfy the above needs, our objectives were: *first*, to predict the performance through analytical models and validate the model predictions; and *second*, to analyze the performance of multithreaded systems and recommend performance related optimizations of the architecture and program workload. We considered three successively detailed multithreaded systems for performance modeling and analysis: a *single processor* system, a *multiprocessor* system, and McGill's EARTH multithreaded multiprocessor system.

We proposed analytical models based on closed queueing networks to predict the performance of multithreaded architectures, developed their solution techniques, and showed their robustness over predictions using open system models. Given program workload and architecture parameters, the models predict performance measures, like the processor uti-

lization, subsystem utilizations, and latency for split-phase operations. We characterized these performance measures using realistic costs of multithreaded operations. We validated the performance models using results from the simulations as well as the EARTH system measurements. We applied the analytical models to optimize the program workload characteristics, as well as to explore different architectural configurations, to achieve performance benefits from multithreading.

First, our analysis of abstract single processor and multiprocessor multithreaded systems showed how the multithreading affects the performance. The duality of processor and memory subsystems can be exploited to yield high processor performance. A concurrent analysis of the network and processor performance brought forth the significance of the network capacity to tune the program workload characteristics to achieve a high processor performance. We showed how the tolerance of latency at a subsystem is affected by program workload characteristics.

Second, our analysis of the EARTH system showed how much performance gains can be achieved under realistic costs of multithreading and subsystem interactions. We presented a solution to the simultaneous possession of the bus at an EARTH node, using only one analytical model. Our model predictions conform to within 5 to 20% of the measurements from actual program executions on the EARTH system. For a compiler/system architect, the performance characterization of the EARTH system showed the gains from multithreading technique. Our results demonstrate the tradeoffs of realistic costs of multithreading on the performance for fine-grain parallel program workload.

Third, we applied the analytical models to the performance optimization on multithreaded systems. We identified the bottlenecks in the EARTH system design, and explored how the changes in different architectural parameters will affect the performance. For example, a *NOW* system with multithreading yields up to a three-fold performance improvement over a single threaded system. Our results also showed that the performance of a multithreaded program workload is more robust as well as closer to optimal than the performance of a single threaded program workload to the changes in data distributions.

The focus of our models is a Single-Program-Multiple-Data (SPMD) computation, since the SPMD model successfully provides users with a tangible set of parameters to characterize the parallel program workloads. We believe that the optimization hints obtained from

an SPMD model provides good heuristics to tune other multithreaded parallel program structures as well.

We have developed a package for an analytical solution of the closed queueing network (CQN) models of multithreaded systems. The solution accounts for simple split-phase transactions among subsystems as well as the simultaneous resource possession problem encountered on the EARTH system.

To validate the performance prediction, we developed measurement tools on the EARTH system. Also, we simulated stochastic petri net models of multithreaded systems to provide an independent verification of our analytical predictions.

Our results provide a strong evidence on the usefulness of performance models to compilers and system architects for performance optimizations on multithreaded systems.

9.2 Future Directions

Analyses in this thesis show that an extensive set of program workload and architectural parameters is needed to characterize a multithreaded program execution on multiprocessor systems. On the architectural aspects, it is necessary to identify a minimal set of primitives to perform all tasks in a multithreaded program execution. Further, timing overheads and design implications of this set of primitives should be studied for multithreaded systems with off-the-shelf components as well as custom-designed components. On the program workload characteristics, for a given multithreaded multiprocessor system, the performance analysis should be used to assist compiler/run-time system in program workload distribution.

In the long term, a compiler or a run-time system should integrate the performance analysis to perform program workload partitioning as well as data-set partitioning to achieve high processor performance. Also, the performance analysis should be used from early design stages of the multithreaded system. This helps to identify and correct potential performance bottlenecks, before major design implementation related decisions are taken.

Bibliography

[1] S. Abraham and K. Padmanabhan. Performance of the direct binary n-cube network for multiprocessors. *IEEE Transactions on Computers*, C-38(7):1000–1011, July 1989.

[2] V.S. Adve and M.K. Vernon. Performance analysis of mesh interconnection networks with deterministic routing. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):225–247. March 1994.

[3] A. Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[4] A. Agarwal. Performace tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[5] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and performance. In *Proc. of the 22nd Int'l. Symp. on Computer Architecture*, pages 2–13, 1995.

[6] A. Agarwal, B.H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proc. of the 17th Int'l. Symp. on Computer Architecture*, pages 104–114, 1990.

[7] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[8] L. Alkalaj and R.V. Bopanna. Performance of multi-threaded execution in a shared-memory multiprocessor. In *Proc. of 3rd Ann. IEEE Symp. on Parallel and Distributed Processing*, pages 330–333, Dallas, USA, December 1991.

[9] R. Alverson et al. The Tera computer system. In *Proc. of the 1990 Int. Conf. on Supercomputing*, pages 1–6, Amsterdam, Netherlands, June 1990. ACM.

[10] Jennifer M. Anderson, Saman P. Amarasinghe, and Monica S. Lam. Data and computation transformations for multiprocessors. In *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 166–178. 1995.

[11] T. Anderson, D. Culler, and D. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, 1995.

[12] Ramune Arlauskas. iPSC/2 system: A second generation hypercube. In *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 38–42, NY, 1988. ACM.

[13] R.H. Arpaci, D.E. Culler, A. Krishnamurthy, S.G. Steinberg, and K. Yellick. Empirical evaluation of the CRAY-T3D: A compiler perspective. In *Proceedings of 22nd Annual International Symposium on Computer Architecture*, pages 320–331. ACM, 1995.

[14] Arvind and R.A. Iannucci. Two fundamental issues in multiprocessing. Computation Structures Group Memo 226, Laboratory for Computer Science, MIT, 1987. Also in Procs. of 4th Int'l. DFVLR Seminar on Foundations of Engineering Sciences, Bonn, Germany, June 1987, pp 61–88.

[15] F. Baskett, K. Mani Chandy, R.R. Muntz, and F.G. Palacios. Open, closed, and mixed network of queues with different classes of customers. *Journal of the ACM*, 22(2):248–260, April 1975.

[16] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216. 1995.

[17] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, pages 29–36, February 1995.

[18] B. Boothe and A. Ranade. Improved multithreading techniques for hiding communication latency in multiprocessor. In *Proc. of the 19th Int'l. Symp. on Computer Architecture*, 1992.

[19] E.L. Boyd and E.S. Davidson. Communication in the KSR1 MPP: Performance evaluation using synthetic workload experiments. In *Proceedings of the ACM 1994 International Conference on Supercomputing, U.K.* 1994.

[20] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat. Latency hiding in message-passing architectures. In *Proceedings of International Parallel Processing Symposium*, pages 704–709, April 1994.

[21] J.P. Buzen. Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM*, 16(9):527–531, September 1973.

[22] D. Chaiken and A. Agarwal. Software-extended coherent shared memory: Performance and cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*. ACM, 1994.

[23] Y-K. Chong and K. Hwang. Performance analysis of four memory consistency models for multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 6(10):1085–1099, October 1995.

[24] D.E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of 15th Annual International Symposium on Computer Architecture*, pages 141–150. ACM, May 1988.

[25] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T.v. Eicken, and K. Yelick. Parallel programming in split-c. In *Supercomputing'93*, Nov 1993.

[26] D.E. Culler, A. Sah, K.E. Schauser, T.v. Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proc. of the 4th Intl. Conference on ASPLOS*, pages 164–175, April 1991.

[27] D.E. Culler et al. Logp- towards a realistic model of parallel computation. In *Proc. of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. May 1993.

[28] Z. Cvetanovic and D. Bhandarkar. Performance characterization of the Alpha 21164 microprocessor using TP and SPEC workloads. In *Proc. of the 2nd Int'l Conference on High Performance Computer Architecture*, pages 270–280, San Jose, CA, February 1996.

[29] Y. Dallery and X.-R. Cao. Operational analysis of stochastic closed queuing networks. *Performance Evaluation*, 14:43–61, 1992.

[30] W. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):775–785, June 1990.

[31] E. de Souza e Silva and R.R. Muntz. Approximate solutions for a class of non-product form queueing network models. *Performance Evaluation*, 7(1987):221–242, 1987.

[32] J.B. Dennis and G.R. Gao. An efficient pipelined dataflow processor architecture. In *Joint Conf. on Supercomputing*, pages 368–373, Florida, Nov. 1988. IEEE Computer Society and ACM SIGARCH.

[33] J.B. Dennis and G.R. Gao. Evolution of multithreaded architectures. In R.A. Iannucci, G.R. Gao, R.H. Halstead, Jr., and B.J. Smith, editors, *Multithreaded Computer Architecture: A Summary of the State of the Art*, chapter 1. Kluwer, Norwell, MA, 1994.

[34] S.J. Eggers and R.H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proc. of the 32rd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 257–270. ACM, April 1989.

[35] R.J. Eickemeyer, R.E. Johnson, S.R. Kunkel, M.S. Squillante, and S. Liu. Evaluation of multithreaded uniprocessors for commercial application environments. In *Proceedings of 23th Annual International Symposium on Computer Architecture*. ACM, 1996.

[36] E.W. Felten and D. McNamee. Improving the performance of message-passing applications by multithreading. In *Proceedings of the 6th Scalable High Performance Computing Conference*. IEEE, 1992.

[37] M. Filio, S.W. Keckler, W.J. Dally, N.P. Carter, A. Chang, Y. Gurevich, and W.S. Lee. The M-Machine Multicomputer. In *Proceedings of the 28th Annual Symposium on Micro-Architecture*, Ann Arbor, Michigan, November 1995.

[38] Gesellschaft für Mathematik und Datenverarbeitung mbH. *MANNA Hardware Reference Manual.* Berlin, Germany, 1993.

[39] Q.S. Gao. The chinese remainder theorem and the prime memory system. In *Proceedings of 20th Annual International Symposium on Computer Architecture*, pages 337–340. ACM, 1993.

[40] R. Govindarajan and S.S. Nemawarkar. Small–a scalable multithreaded architecture. In *Proceedings of the 4th International Symposium on Parallel and Distributed Processing*, Arlington, Texas, December 1992. IEEE Press.

[41] R.H. Halstead Jr. and T. Fujita. Masa: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of 15th Annual International Symposium on Computer Architecture*, pages 443–451. ACM, June 1988.

[42] M. Heinrich et al. the performance impact of flexibility in the stanford flash multiprocessor. In *Proc. of the 6th Intl. Conference on ASPLOS*, pages 274–285, Oct. 1994.

[43] W.D. Hillis and G.L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, Dec. 1986.

[44] M.A. Holliday and M.K. Vernon. A generalized timed petri net model for performance analysis. *IEEE Transactions on Software Engineering*, SE-13(12):1297–1310, December 1987.

[45] C.-T. Hsieh and S.S. Lam. PAM– a noniterative approximation solution method for closed multichain queueing networks. In *Proceedings of the ACM SIGMETRICS Conference 1988*, pages 261–269. 1988.

[46] H. Hum et al. A design study of the EARTH multiprocessor. In *Proceedings of the 3rd International Conference on Parallel Architectures and Compilation Techniques, PACT-95*, pages 59–68, Cyprus, July 1995.

[47] R.A. Iannucci. *A Dataflow/von Neumann Hybrid Architecture.* PhD thesis, MIT Dept of Electrical Engineering and Computer Science, May 1988.

[48] P.A. Jacobson and E.D. Lazowska. Analyzing queueing networks with simultaneous resource possession. *Communications of the ACM*, 25(2):142–151, February 1982.

[49] R. Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation and modeling.* Wiley, New York, 1991.

[50] K. Johnson. The impact of communication locality on large-scale multiprocessor performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 392–402. ACM, May 1992.

[51] S.W. Keckler and W.J. Dally. Processor Coupling: Integrating Compile Time and Runtime Scheduling. In *Proceedings of the 19th International Symposium on Computer Architecture.* ACM, May 1992.

[52] L. Kleinrock. *Queueing Systems*, volume II. John Wiley & Sons, Inc., USA, 1976.

[53] C.P. Kruskal and M. Snir. The performance of multistage interconnection networks. *IEEE Transactions on Computers*, C-32(12):1091–1098, Jan 1983.

[54] K. Kurihara, D. Chaiken, and A. Agarwal. Latency tolerance in large-scale multiprocessors. In *Proceedings of the 19th International Symposium on Shared Memory Multiprocessing.* ACM, 1991.

[55] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessor and workstations. In *Proc. of the 6th Intl. Conference on ASPLOS*, pages 308–318, Oct. 1994.

[56] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

[57] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of 19th Annual International Symposium on Computer Architecture*, pages 92–103. ACM, 1992.

[58] S.T. Leutenegger and M.K. Vernon. A mean-value performance analysis of a new multiprocessor architecture. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 167–176. 1988.

[59] O. Maquelin, H.H.J. Hum, and G.R. Gao. Costs and benefits of multithreading with off-the-shelf RISC processors. In *Procs. EURO-PAR'95*. Springer-Verlag, 1995.

[60] A. M. Marsan, G. Balbo, and G. Conte. A class of generalized stochastic petri nets for the performance analysis of multiprocessor systems. *ACM Transactions on Computer Systems*, 4(2):93–122, 1984.

[61] M. Misra. *IBM RS System/6000 Technology, First Edition*. IBM, Austin, 1990.

[62] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[63] S.S. Nemawarkar and G.R. Gao. Performance analysis of multithreaded multiprocessors using an integrated system model. ACAPS Technical Memo 84-1, School of Computer Science, McGill University, Montreal, Que., April 1995. In revision for *Parallel Computing* journal.

[64] S.S. Nemawarkar and G.R. Gao. Measurement and modeling of earth-manna multithreaded architecture. In *Proceedings of the 4th International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'96)*, pages 109–114. IEEE, 1996.

[65] S.S. Nemawarkar, R. Govindarajan, G.R. Gao, and V.K. Agarwal. Performance evaluation of latency tolerant architectures. In *Proceedings of the 4th International Conference on Computing and Information*, pages 183–186, Toronto, Canada, May 1992. IEEE Press.

[66] S.S. Nemawarkar, R. Govindarajan, G.R. Gao, and V.K. Agarwal. Analysis of multithreaded multiprocessors with distributed shared memory. In *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, pages 114–121, Dallas, Texas, December 1993.

[67] S.S. Nemawarkar, R. Govindrajan, G.R. Gao, and V.K. Agarwal. Performance of interconnection network in multithreaded architectures. In *Proceedings of Parallel Architectures and Languages Europe (PARLE)*, pages 823-826. Springer-Verlag, LNCS 817, Athens, Greece, July 1994.

[68] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Procs. of 19th Annual International Symposium on Computer Architecture*, pages 156–167. ACM, 1992.

[69] Q. Ning, V. van Dongen, and G.R. Gao. Automatic data and computation decomposition for distributed memory machines. ACAPS Technical Memo 82, School of Computer Science, McGill University, Montreal, Que., March 1994.

[70] M.D. Noakes, D.A. Wallach, and W.J. Dally. The j-machine multicomputer: An architecture evaluation. In *Proceedings of 20th Annual International Symposium on Computer Architecture*, pages 224–235. ACM, 1993.

[71] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing'95*, Dec 1995.

[72] P. Parent and O. Tanir. Voltaire: a discrete event simulator. In *Proceedings of Fourth International Workshop on Petri Nets and Performance Models*, Melbourne, Australia, December 1991.

[73] Steve A. Przybylski. *Cache and Memory Hierarchy Design: A Performance-Directed Approach.* Morgan Kaufman, 1990.

[74] B. Ramakrishna Rau. Pseudo-randomly interleaved memory. In *Proceedings of 18th Annual International Symposium on Computer Architecture*, pages 74–83. ACM, 1991.

[75] M. Reiser and S. Lavenberg. Mean value analysis of closed multichain queueing networks. *Journal of ACM*, 27(2):313–322, April 1980.

[76] K.A. Robbins and S. Robbins. *The Cray X-MP/Model 24 : a case study in pipelined architecture and vector processing*, volume 374 of *LNCS*. Springer-Verlag, 1989.

[77] L. Roh, W.A. Najjar, B. Shankar, and A.P.W. Bohm. An evaluation of optimized threaded code generation. In *Proceedings of the 2nd International Conference on Parallel Architectures and Compilation Techniques, PACT-94*, pages 37–46, Montreal, August 1994.

[78] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. In *Proceedings of 20th Annual International Symposium on Computer Architecture*, pages 14–25. ACM, 1993.

[79] R.H. Saavedra, W. Mao, and K. Hwang. Performance and optimization of data prefetching strategies in scalable multiprocessors. *Journal of Parallel and Distributed Computing*, 22:427–448, 1994.

[80] R.H. Saavedra-Barrera, D.E. Culler, and T. von Eicken. Analysis of multithreaded architectures for parallel computing. In *Proc. of 2nd Ann. ACM Symp. on Parallel Algorithms and Architectures*, Crete, Greece, July 1990.

[81] R.H. Saavedra-Barrera, D.E. Culler, and T. von Eicken. Analysis of multithreaded architectures for parallel computing. In *Proc. of 2nd Ann. ACM Symp. on Parallel Algorithms and Architectures*, Crete, Greece, July 1990.

[82] S. Sakai, K. Okamoto, H. Matsuoka, H. Hirono, Y. Kodama, and M. Sato. Super-threading: Architectural and software mechanisms for optimizing parallel computation. In *Proceedings of the ACM 1993 International Conference on Supercomputing*, pages 251–260. July 1993.

[83] Mitsuhisa Sato, Yuetsu Kodama, Shuichi Sakai, and Yoshinori Yamaguchi. EM-C: Programming with explicit parallelism and locality for the EM-4 multiprocessor. In *Proceedings of the 2nd International Conference on Parallel Architectures and Compilation Techniques, PACT-94*, pages 3–14, Montreal, August 1994.

[84] K.E. Schauser, D.E. Culler, and S.C. Goldstein. Sequential constraint paritioning– a new algorithm for partitioning non-strict programs into sequential threads. In *Proc. of ACM Symposium on Principles of Programming Languages*. Jan. 1995.

[85] Klaus Eric Schauser, David E. Culler, and Thorsten von Eicken. Compiler-controlled multithreading for lenient parallel languages. Technical Report UCB/CSD 91/640, University of California, Berkeley, 1991.

[86] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[87] B. Smith. The architecture of HEP. In J.S. Kowalik, editor, *Parallel MIMD Computation: HEP Supercomputer and its Application*, pages 41–55. The MIT Press, 1985.

[88] J.E. Smith and W.R. Taylor. Accurate modeling of interconnection networks in vector supercomputers. In *Proceedings of the ACM 1991 International Conference on Supercomputing, Netherlands*, pages 264–273. 1991.

[89] A. Sohn, M. Sato, N. Yoo, and J-L. Gaudiot. Effects of multithreading on data and workload distribution for distributed-memory multiprocessors. In *Proceedings of International Parallel Processing Symposium*, April 1996.

[90] R. Thekkath and S. Eggers. The effectiveness of multiple hardware contexts. In *Proc. of the 6th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 328–337. ACM, Oct. 1994.

[91] R. Thekkath and S. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *Proceedings of the 21st International Symposium on Computer Architecture*. ACM, April 1994.

[92] K.B. Theobald. SEMi: A simulator for EARTH, MANNA, and the i860 (version 0.10). ACAPS Technical Note 43, School of Computer Science, McGill University, Montreal, Que., April 1996.

[93] M.R. Thistle and B.J. Smith. A processor architecture for Horizon. In *Supercomputing '88*, Orlando, Florida, Nov 1988.

[94] J. Torrellas, J. Hennessy, and T. Weil. Analysis of critical architectural and program parameters in a hierarchical shared-memory multiprocessor. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 163–172. 1990.

[95] K.S. Trivedi. *Probability and statistics with reliability, queuing, and computer science applications*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[96] K.S. Trivedi, B.W. Haverkort, A. Rindos, and V. Mainkar. Techniques and tools for reliability and performance evaluation: Problems and perspectives. In *Procs. of the 7th International Conferenec on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 1–24, Vienna, Austria, 1994. Springer-Verlag, LNCS-794.

[97] B. VanVoorst, S. Seidel, and E. Barszcz. Profiling the communication workload of an ipsc/860. In *Proceedings of the 8th Scalable High Performance Computing Conference*, pages 221–228. IEEE, May 1994.

[98] C.A. Waldspurger and W.E. Weihl. Register relocation: Flexible contexts for multithreading. In *Proceedings of the 20th International Symposium on Computer Architecture*. ACM, May 1993.

[99] W.-D. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proc. of the 3rd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 243-256. ACM, 1989.

[100] W.D. Weber and A. Gupta. Exploring the benefits of multiple contexts in a multiprocessor architecture: Preliminary results. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 273–280. ACM, 1989.

[101] D. Willick and D. Eager. An analytic model of multistage interconnection networks. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 192–202. 1990.

[102] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of 22nd Annual International Symposium on Computer Architecture*, pages 24–36. ACM, 1995.

[103] W. Yamamoto, M.J. Serrano, A.R. Talcott, R.C. Wood, and M. Nemirovsky. Performance estimation of multistreamed, superscalar processors. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, pages 194–204, 1994.

[104] K.C. Yeager. Mips R10000 superscalar microprocessor. *IEEE Micro*, pages 28–40, April 1996.

Appendix A

Approximate Mean Value Analysis

In this section, we describe the assumptions for queueing networks such that they have *product-form* solutions [15, 75, 56, 49]. Next, we describe an approximate mean value analysis (AMVA) to solve such queueing network models in this thesis.

A.1 Assumptions for Product-Form Solution

A queueing network is called a *product-form* network if the equilibrium probability of the state of the queueing network is a product of functions of queue lengths at each functional unit. For a queueing network with M nodes, let n_i be the queue length at node i and $f_i(n_i)$ be a function of queue length n_i. Let N be the total number of accesses in the system. Then the equilibrium state probability P is given as follows:

$$P(n_1, n_2, ..., n_M) = \frac{1}{G(N)} \prod_{i=1}^{M} f_i(n_i) \tag{A.1}$$

where $G(N)$ is a normalizing constant and a function of N.

The significance of *product-form* networks is that to obtain the equilibrium probability of a state we do not need to enumerate all states of the system. Thus, large systems can be analyzed easily. Based on this concept, many computationally efficient techniques have been developed, e.g. Mean value analysis [75], Approximate mean value analysis [75, 56], and Convolution algorithm [21].

For a queueing network to have a product form solution, following assumptions should be satisfied [15, 56]:

1. *Service Disciplines*: All service centers have one of the following four types of service disciplines: First Come, First Served (FCFS), Processor Sharing (PS), Infinite Servers (IS), and Last Come, First Served Preemptive Resume (LCFS-PR).

2. *Job Classes*: The accesses belong to a single class while awaiting or receiving service at a service center but may change classes and service centers according to fixed probabilities at the completion of a service request.

3. *Service Time Distributions*: At FCFS service centers, the service time distributions must be identical and exponential for all classes of jobs. At other service centers, where the service times should have probability distributions with rational Laplace transforms, different classes of jobs may have different distributions.

4. *State-dependent Service*: The service time at a FCFS service center can depend only on the total queue length of the center. The service time for a class at PS, LCFS-PR, and IS centers can also depend on the queue length for that class, but not on the queue length of other classes. Moreover, the overall service rate of a subnetwork can depend on the total number of jobs in the subnetwork.

5. *Arrival Processes*: In open networks, the time between successive arrivals of a class should be exponentially distributed. No bulk arrivals are permitted. The arrival rates may be state dependent. A network may be open with respect to some classes of jobs and closed with respect to other classes of jobs.

6. *Job Flow Balance*: For each class, the number of arrivals to a device must equal the number of departures from the device.

7. *One-Step Behavior*: A state change can result only from single accesses entering the system, moving between pairs of devices in the system, or exiting the system. This assumption asserts that simultaneous access moves will not be observed.

8. *Device Homogeneity*: A device's service rate for a particular class does not depend on the state of the system in any way except for the total device queue length and the designated class's queue length. This assumption translates to the following:

(a) *Single-Resource Possession*: An access may not be present (waiting for service or receiving service) at two or more devices at the same time.

(b) *No Blocking*: A device renders service whenever accesses are present; its ability to render service is not controlled by any other device.

(c) *Independent Job Behavior*: Interaction among accesses is limited to queueing for physical devices; for example there should not be any synchronization requirements.

(d) *Local Information*: A device's service rate depends only on local queue length and not on the state of the rest of the system.

(e) *Fair Service*: If service rates differ by class, the service rate for a class depends only on the queue length of that class at the device and not on the queue lengths of other classes. This means that the servers do not discriminate against accesses in a class depending on the queue lengths of other classes.

(f) *Routing Homogeneity*: The access routing should be state independent.

A.2 AMVA Algorithm

Now we discuss the *approximate mean value analysis* (AMVA) to solve the closed queueing networks which have a *product form* solution. This AMVA algorithm has been excerpted from Lazowska *et al* [56], and suitably modified for our work. We assume the same terminology as described earlier in Section 5.2 and the following Table B.1.

Inputs to the AMVA are architectural parameters and program workload parameters. We compute service demands, $\rho_{i,m}$, at each service center for each class of accesses. The AMVA considers two population vectors, representing the number of threads for each class of accesses. The vector $(\mathbf{N}) = (n_t, ..., n_t)$ indicates that n_t threads can be executed by each processor. The population vector $(\mathbf{N} - \mathbf{1}_i)$ indicates that processor i has $(n_t - 1)$ threads, while other processors have n_t threads each.

The AMVA computes (see Figure A.1):

1. the arrival rate λ_i for the threads belonging to each processor i;

2. the waiting time $w^*_{i,m}$ at each node m; and

3. the queue length $n^*_{i,m}$.

Statistics for population vector $(\mathbf{N} - \mathbf{1}_i)$ are used to compute the queue lengths and waiting times for n_t-th thread added on to the processor i. Statistics for population vector $\mathbf{N}$ indicate the steady state performance measures when n_t threads are present at each processor.

The AMVA uses statistics at populations $(\mathbf{N} - \mathbf{1}_i)$ and $(\mathbf{N})$. The intuition is that a newly added thread to a class (i.e. new population $\mathbf{N}$) *sees* the queueing network in equilibrium with respect to the population $(\mathbf{N} - \mathbf{1}_i)$. *First*, using the queue lengths at each service node for population $(\mathbf{N} - \mathbf{1}_i)$, waiting times are computed for the new thread (access). *Second*, waiting times for each class are used to compute throughputs. *Third*, using Little's law [56], queue lengths at each service nodes are computed. Thus performance measures of interest are obtained at the population $(\mathbf{N})$. Figure A.1 shows 5 steps in the AMVA algorithm.

Step 1 in the AMVA is an initial guess for performance measures when the thread population in the system is $(\mathbf{N})$. These performance measures are the queue lengths at each of M nodes, for P classes of threads. Note that the thread population in each class i, i.e. $N_i(= n_t)$, is equally distributed on M nodes. The speed of convergence of Steps 2 to 5 depends on the closeness of this guess to the final queue length distribution [45].

Step 2 computes waiting time $w^*_{i,m}$ for class i access at node m. Step 2(a) obtains the queue length $n^*_{i,m}$ at each node m when class i population is $(N_i - 1)$ i.e. the population vector $(\mathbf{N} - \mathbf{1}_i)$. The first term "1" represents the newly arrived thread (or access) for class i. The second term uses an *approximation* function to compute queue lengths of class i accesses. The approximation is an *interpolation* of $n^*_{i,m}(\mathbf{N})$, that is, $\frac{N_i-1}{N_i} n^*_{i,m}(\mathbf{N})$. The third term is the queue length for other classes. Since population of other classes does not change, the queue length (i.e. the number of accesses at each node) for these classes remains the same. Step 2(b) computes the waiting time for the class i access using the service demand at node m and queue length at node m for class i.

In Step 3, the *cycle time for a thread in class i* is a sum of waiting times at all nodes for a newly added thread in class i. Hence, the throughput for class i is $\frac{n_t}{\textit{cycle time for one thread}}$.

Step 4 uses Little's law. New queue lengths are computed at each node m and for each class i, using the throughput for class i and the waiting time for a class i access at node m.

Finally Step 5 verifies whether the maximum difference between queue lengths from successive iterations are within the tolerance level. Iterations for Steps 2 to 5 continue till the maximum difference is acceptable.

1- Initialize $\quad n^*_{i,m}(\mathbf{N}) = \frac{N_i}{M}$

2- Compute at each node m and for each class i

a- $\quad n^*_{i,m}(\mathbf{N} - \mathbf{1}_i) = 1 + \left[\frac{N_i - 1}{N_i} n^*_{i,m}(\mathbf{N})\right] + \sum_{j=1, j\neq i}^{P} n^*_{j,m}(\mathbf{N})$

b- $\quad w^*_{i,m}(\mathbf{N}) = \rho_{i,m}\left[n^*_{i,m}(\mathbf{N} - \mathbf{1}_i)\right]$

3- Compute for each class i

$$\lambda_i(\mathbf{N}) = \frac{N_i}{\sum_{m=1}^{M} w^*_{i,m}}$$

4- Compute new values for $n^*_{i,m}(\mathbf{N})$ at all nodes m and for all classes i

$$n^*_{i,m}(\mathbf{N}) = \lambda_i(\mathbf{N}) w^*_{i,m}(\mathbf{N})$$

5- If $difference(n^*_{i,m}(\mathbf{N})_{new},\ n^*_{i,m}(\mathbf{N})_{old}) > tolerance$
then go to *step 2* else *exit*

Figure A.1: AMVA Algorithm

Appendix B

Symbols and Their Meanings

Workload Parameters	
n_t	Number of threads at each processor
$R = \frac{1}{\mu_p}$	Mean value of thread runlength
p_{remote}	Probability of accessing a remote memory module
rw	Number of Read/Writes in the duration R
p_{sw}	Probability that a message is sent to neighboring switch
-	# and type of long latency operations— GET_SYNC, BLKMOV

System Parameters (*Values measured on EARTH*)		
C	Context switch overhead (END_THREAD, Scheduling etc.)	37 cycles
$L = \frac{1}{\mu_m}$	Memory latency for each access	10 cycles
S	Routing delay at a network switch	
k	Number of processors in one dimension, i.e., row/column	
SU_{serv}	SU processing time for each access (other than BLKMOV)	20 cycles
$lnkin_{serv}$	Link access time for incoming network message	15 cycles
$lnkout_{serv}$	Link access time for sending network message	8 cycles
xin_{serv}	Delay at input port of network switch	8 cycles
$xout_{serv}$	Routing delay at output port of network switch	32 cycles
n_p	Number of ports at the memory	1
$P = k^2$	Number of nodes in the system	2 to 16

Output Parameters	
L_{obs}	Observed memory latency (with queuing delay)
S_{obs}	Observed network latency (individual message type)
$L_{get-sync}$	Latency for GET_SYNC operation. *ded* subscript ⇒ dedicated node
λ_{net}	Message rate from processor to the IN (message type)
U_p	Processor (EU) utilization
U_{net}, U_{sw}, ρ	Switch Utilization
U_m	Memory utilization
U_{sys}	System utilization, and PSU is the peak system utilization
LSI_{n_t}	Locality sensitivity index for n_t threads
$MLSI_{P,n_t}$	Multiprocessor locality sensitivity index for program on P nodes

Analysis Related and Derived Parameters	
xm	Number of accesses at the memory subsystem
G	Normalization constant for product-form solution
$em_{i,j}$	Visit ratio of thread from processor i to memory module j
$e_{i,j,fu}$	Visit ratio of thread from processor i to functional unit fu at j
$\rho_{i,m}$	Visit ratio for thread from processor i at node m × service rate at node m
$\rho_{i,j,fu}$	Service demand ρ for thread of class i at functional unit fu of node j
M	Number of nodes in the system
N	Maximum thread population, n_t, at each processor in the system (for AMVA)
$N-1_i$	Processor i having n_t-1 threads while the rest have n_t threads
$n^*_{i,m}$	Queue length at node m, due to threads from processor i
λ_i	Message arrival rate at the processor i
$w^*_{i,m}$	Waiting time (including service time) for thread from processor i at node m
$w^*_{i,j,fu}$	Waiting time for thread from processor i functional unit fu at node j
h	Number of hops traveled by a message from its source node
a	Normalization constant for geometric distribution
d_{avg}	Average distance (in hops) for a message on the network
k_d	Average distance (in hops) for a message in a dimension
B	Bandwidth of the network
D_P	Data distribution on P processing nodes
T_{P,n_t}	Execution time with n_t threads on P processing nodes

Table B.1: Model Parameters

Appendix C

Throughput of Pipelined Networks

The objective of this section is to determine the performance of a pipelined 2-dimensional mesh network under saturation.

We consider the following problems for k-ary n-cube networks, with an emphasis on 2-dimensional meshes:

Problem C.0.1 *For a 2-dimensional open network, under an access pattern, what is the (analytical) value of the maximum throughput?*

Problem C.0.2 *For a network topology (say, a 2-dimensional mesh), what is the maximum throughput of an open network under any (source-destination) access pattern?*

Problem C.0.3 *For above 2-dimensional network embedded in the multithreaded multiprocessor system, under the same access pattern as in* Problem C.0.1, *what is the (analytical) value of the maximum throughput? Do the values in Problems C.0.1 and C.0.3 match?*

The throughputs are derived for pipelined k-ary, n-cube networks. k is the number of processing nodes in each of the n dimensions of the network. Here the term *pipelined network* refers to a *worm-hole routed* network, which allows a switch to begin forwarding of a message as soon as the message header arrives (in absence of contentions) [30, 3]. The

remote access pattern is assumed to be the same for each processor. A *channel* or *link* represents a connection between two adjacent switches on the network. For example, a switch node in a 2-dimensional mesh network has 4 channels. The terminology used here is consistent with the one used by Agarwal [3]:

n is the number of dimensions of the k-ary, n-cube network.

m is the probability of a network request from a processing node.

ρ is the probability of a unit-sized message arriving at the channel. ρ is also same as the channel utilization.

U_{sw} is the utilization of the switch node. U_{sw} is an average utilization of the channels connected to the switch.

k_d is the average distance in each dimension. Thus, $k_d = \frac{k}{4}$, if k is even and channel is bidirectional.

d_{avg} is the average distance travelled by a message on the network. Thus, $d_{avg} = n\ k_d$, and $d_{avg} = 2\ k_d$, when $n = 2$.

B is the size of a messages in flits, i.e. the time it takes to get service at a switch.

S is the delay for a message at a switch on a store-and-forward network, which is not pipelined. The value of S in a store-and-forward network is the same as B in a pipelined network.

λ_{net} is the number of remote messages sent by the processor in each cycle. Given an access pattern, $\lambda_{net,saturation}$ is the maximum value of λ_{net}.

Now, let us consider the first problem.

Problem C.0.1: *For a 2-dimensional open network, under an access pattern, what is the analytical value of the maximum throughput?* [1]

Let us consider a switch node with links in n dimensions. The switch receives m network messages per cycle from the processing node attached to it. A message travels a distance of

[1]Note that this is the saturation value of "actual/achieved traffic" when "offered traffic" is increased indefinitely.

$d_{avg}(= n\ k_d)$ hops. In other words, $m\ n\ k_d$ messages from $n\ k_d$ nodes are at a switch node, in each cycle. Dividing this traffic at a switch among the $2n$-channels we get the channel utilization:

$$\rho = \frac{m\ n\ k_d}{2n} = m\ k_d/2 \tag{C.1}$$

In Equation C.1, the numerator $m\ n\ k_d$ is the total number of messages arriving/leaving a switch node, and the denominator is the number of links used for transferring these messages. When we assume that all links carry equal load, d_{avg} and k_d are the only parameters affected by a particular access pattern. For a 2-dimensional network, there are 4 links connected to each switch. The utilization at a switch, U_{sw}, is an average of utilization of these links.

$$U_{sw} = 2n\ \rho = m\ n\ k_d \tag{C.2}$$

$$= m\ d_{avg} \tag{C.3}$$

A switch saturates when U_{sw} equals "1". We obtain the throughput for a switch using Equation C.3.

$$m = \frac{1}{d_{avg}} \quad \text{for separate channels in each direction.} \tag{C.4}$$

$$= \frac{1}{2\ d_{avg}} \quad \text{for bidirectional channels at a switch.} \tag{C.5}$$

Thus, the maximum throughput is a reciprocal of the number of switch nodes a message travels.

When the size of each message is B flits, the service time for this message at a switch requires B cycles. In other words, the effect of increasing the packet size to B flits can be approximated by increasing the delay through the switch by a factor B to reflect the increase in the service time of each packet [3, 53]. Reflecting the increased packet size in the Equation C.1, we get:

$$\rho = \frac{m\ B\ n\ k_d}{2n} = \frac{m\ B\ d_{avg}}{2n} \tag{C.6}$$

Equation C.6 can be interpreted as follows. A channel receives $\frac{m}{n}$ messages per cycle through a switch (from the processing node attached to it). Every message keeps each of d_{avg} channels busy for B cycles each. That is, each of m messages from d_{avg} processing nodes keeps a channel busy for B cycles.

For a 2-dimensional mesh, the switch utilization U_{sw} for messages of size B flits is given by:

$$U_{sw} = 2n\ \rho \ = m\ B\ d_{avg} \quad \text{for separate channels in each direction.} \tag{C.7}$$

$$U_{sw} \ = \frac{m\ B\ d_{avg}}{2} \quad \text{for bidirectional channels.} \tag{C.8}$$

Equations C.7 and C.8 show that a message occupies a switch and associated channel for B cycles. Messages from d_{avg} processing nodes arrive at a switch. In a cycle, m messages are sent to the network by a processing node. The maximum throughput, m, is the reciprocal of the duration for which these messages occupy the switch. That is,

$$m \ = \frac{1}{B\ d_{avg}} \quad \text{for separate channels in each direction.} \tag{C.9}$$

$$m \ = \frac{1}{2\ B\ d_{avg}} \quad \text{for bidirectional channels.} \tag{C.10}$$

Now, we compute the maximum throughput of an open network.

Problem C.0.2: *For a network topology (say, a 2-dimensional mesh), what is the maximum throughput of an open network under any (source-destination) access pattern?*

For a 2-dimensional mesh network (and k-ary,n-cube, in general), an ideal mapping requires only a single network hop. Also, the messages are sent to all neighbors at a distance of one network hop, i.e., an average distance in each dimension, k_d, is 1. Using Equations C.4 and C.9, we get

$$m \ = 1 \quad \text{for unit-sized messages} \tag{C.11}$$

$$= \frac{1}{B} \quad \text{for } B\text{-flit messages} \tag{C.12}$$

Now let us compute above throughputs when the network is embedded in a multiprocessor system, which is executing a parallel program workload.

Problem C.0.3: *For above 2-dimensional network embedded in the multithreaded multiprocessor system, under the access pattern in Problem C.0.1, what is the analytical value of the maximum throughput?*

The interaction between the network and the rest of the system occurs through the interface between the switch and the processing node. The number of messages per cycle, sent by a processing node to the attached switch node, is m. Without a loss of generality, we

assume that for each message sent by a processor, the memory at a remote node responds with one message. Thus, a processor sends $\frac{m}{2}$ messages per cycle, and the memory sends the rest of $\frac{m}{2}$ messages per cycle. Since λ_{net} is the message rate from the processor to the network, λ_{net} equals $\frac{m}{2}$. Also, the message size B is same as the switch delay S. From Equations C.9 and C.10, we obtain the saturation value of λ_{net}.

$$\lambda_{net,saturation} = \frac{m}{2} = \frac{1}{2\ d_{avg}\ B} = \frac{1}{2\ d_{avg}\ S} \quad \text{for separate channels in each direction.} \tag{C.13}$$

$$\lambda_{net,saturation} = \frac{m}{2} = \frac{1}{4\ d_{avg}\ B} = \frac{1}{4\ d_{avg}\ S} \quad \text{for bidirectional channels.} \tag{C.14}$$

Equation C.13 is same as the network *capacity*(Section 5.6). Thus, the maximum throughput of a network in a multiprocessor system is the same irrespective of whether the network is pipelined.

Appendix D

McGill EARTH-MANNA System

The EARTH (an Efficient Architecture for Running Threads) architecture proposes an efficient execution of the synchronization operations and the computations using different functional units [46]. Currently, the EARTH programming model is implemented on the MANNA multiprocessor, developed at GMD FIRST, Germany. The structure of a MANNA node is close to the EARTH node architecture, thus helping an efficient emulation of the EARTH architecture through the EARTH-MANNA run-time system and the MANNA hardware. The EARTH Threaded-C compiler offers direct support for EARTH operations, expanding them inline in order to reduce their overhead to a minimum.

EARTH Architecture: An EARTH multiprocessor system consists of multiple EARTH nodes and an interconnection network. Each EARTH node consists of an Execution Unit (EU) and a Synchronization Unit (SU), linked by a pair of buffers (see Figure 7.2). The SU and EU share a local memory, which is part of a distributed shared memory.

The processing functions at an EARTH node are distributed onto two units: the EU executes the application program code, and the SU performs the synchronization and communication operations. The EU processes instructions in an *active* thread, where an *active thread* is initiated for execution when the EU fetches its *thread id* from the *ready queue*. The EU executes a thread to completion before moving to another thread. It interacts with the SU and the network by placing messages in the *event queue*. The SU fetches these messages from the *event queue*, in addition to the messages coming from remote processors through the network. The SU responds to remote synchronization commands and requests

for data, and also determines which threads are to be run and adds their thread ids to the ready queue.

MANNA System: The MANNA (Massively parallel Architecture for Non-numerical and Numerical Applications) multiprocessor system consists of multiple high performance MANNA nodes connected to the leaves of a high bandwidth interconnection network. Each MANNA node consists of two Intel i860 XP RISC processors clocked at 50 MHz, 32 MB of dynamic RAM and a bidirectional network interface (see Figure 7.2). The link interface is capable of transferring 50 MB/S in each direction simultaneously, for a total bandwidth of 100 MB/S per node. The network is based on 16 x 16 crossbar chips which support the full 50 MB/S link speed. Smallest machine configuration is a two-node MANNA-PC with the links directly connected. All higher configurations use crossbar chips (see Figure 7.1). Configurations up to 40-node machine with 4 crossbar chips are in use.

EARTH-MANNA Run-time System: The EARTH node architecture is mapped onto a MANNA node as follows. The EU tasks are performed by one of the processors. The SU tasks are performed by other processor. The *ready queue* and the *event queue* which interface the EU and the SU, are implemented in the local memory. The link interface chips maintain the buffers for the network interface. The capacity of network interface buffers is augmented through the overflow queues maintained in local memory.

EARTH Threaded-C Language: The EARTH program model is implemented as an extension to the C language. These multithreading extensions are the support for, a declaration of threaded functions, the specification of threads within these functions, and the specification of EARTH operations. This explicitly parallel language, called EARTH Threaded-C, allows a programmer to directly specify the partitioning into threads, and the EARTH operations (s)he wants to use. Appendix E provides the details.

Appendix E

Threaded-C Language Extensions

Following is a synopsis of some of the multithreading macros used in this report:

`void INVOKE (int proc_num, proc *fun, params...);`

This macro starts a function on an arbitrary processor. The calling thread is *not* suspended. The parameters to INVOKE are a processor number, the function name, and its parameters.

`void CALL (proc *fun, params...);`

This macro executes a threaded function directly, without going through the invoke mechanism. The calling thread is suspended until the function returns. The parameters to CALL() are the function name and its parameters.

`void END_FUNCTION ();`

This must be the last statement of a threaded function. It must be used even if the function contains only a single thread.

`void RETURN ();`

This statement must be used at the end of a function instead of END_FUNCTION if the function is to be called with the CALL() macro.

`THREAD_nnn:`

(nnn is an integer constant.) Labels with this format indicate the start of a new thread.

`void END_THREAD ();`

This macro signals the end of a thread. This macro will normally be followed by a thread label.

`SLOTS SYNC_SLOTS [nnn];`

This macro reserves space for nnn synchronization slots. This must be the first variable declaration of a function (if sync slots are used).

`void INIT_SYNC (int slot_no, int cnt, int rst, int th_no);`

This macro initializes a synchronization slot. A thread label with the corresponding name must exist for each number used in INIT_SYNC.

Following macros provide synchronization and data-transfer operations across threads, and if required, nodes:

`void RSPAWN (char *fp, char *ip);`

This macro spawns the specified thread. This macro allows the specification of threads that are not local to the current function.

`void RSYNC (SLOT *slot_addr);`

This macro signals the specified sync slot, possibly starting a thread.

`void GET_RSYNC_D (double *src, double *dest, SLOT *slot_adr);`

This macro implements remote loads and start the specified thread after completion of synchronization requirements through `slot_adr`. Similar macros exist for character, integer, and floating point data. `GET_SYNC_x` does the same operation but with the slot specified by its number.

`void DATA_SYNC_D (double val, double *dest, int slot_no);`

This macro implements remote stores to the destination address, and the update of the specified synchronization slot at completion. DATA_RSYNC_x, but with the slot specified by its address.

```
void BLKMOV_RSYNC (char *src, char *dest, long bsize, SLOT *slot_adr);
```

This macro implements block transfers. The source and destination addresses can be located on an arbitrary node. There are no alignment restrictions, but the operation is performed more efficiently if the data is aligned on quad-word (16 bytes) boundaries. The size of the block to transfer is specified in bytes (as in sizeof()).

Appendix F

The MVA Pseudo Code

```
1001: Main Steps
1002: { while (all sets of inputs are analyzed) do
1003:   { Get the next set of inputs;
1004:     Initialize the input parameters;
1005:     Compute Memory_Visit_Ratios for each class in the Closed Queueing Network;
1006:     Compute Switch_Visit_Ratios for each class at each node in the CQN;
1007:     Compute Service_Demands at each node;
1008:     if (simultaneous resource possession)
1009:         { Invoke AMVA_SRP for the EARTH related heuristics;}
1010:     else
1011:         { Invoke AMVA for product_form CQNs;}
1012:     Compute Performance_Measures;
1013:   }
1014: }
```

101: **Memory_Visit_Ratios**
102: { **for** all classes: $class = 1$ to P
103: **for** all processing elements: $node = 1$ to P
104: **if** (local $node$ for a $class$)
105: { $visit\_ratio[class][processor(node)] = 1.0$;
106: $visit\_ratio[class][memory(node)] = 1.0 - p_{remote}$; }
107: **else** (the $node$ is remote)
108: { $visit\_ratio[class][processor(node)] = 0.0$; /* no access to processor */
109: **switch** (memory_distribution)
110: { **case** uniform:
111: $visit\_ratio[class][memory(node)] = \frac{p\_remote}{P-1}$;
112: **case** geometric:
113: $visit\_ratio[class][memory(node)] = \frac{(p\_remote * geometric\_dist(p\_sw, class, node))}{geometric\_normal}$;}
114: }
199: }
201: **Switch_Visit_Ratios**
202: { **Sort** the nodes according to their distance from node (0,0).
203: Place their "id"s in the 2-dimensional array "sorted_pes[distance][ids]".
204: /* **Compute** visit ratio for class "1" as follows */
205: **for** $h = d_{max}$ downto 0
206: { **for** each processor_id k at distance h in sorted_pes[h][ids]
207: { $k \equiv (x, y)$ in two-dimensional mesh
208: { **for** each $neighbor$ of k in {(-1,0), (1,0), (0,-1) and (0,1)} directions
209: **if** ($neighbor$ is more distant than k)
210: {$visit\_ratio$[1][$inbound(k)$] + =
211: $visit\_ratio$[1][$inbound(neighbor)$] + $visit\_ratio$[1][$outbound(neighbor)$];}
212: }
213: }
214: }
215: **Repeat** similar computation for other classes;
216: **if** (the access pattern is same), obtain new visit ratios using proper offsets;

```
301: Service_Demands
302: { for all classes: i = 1 to P
303:     { for all processing elements: j = 1 to P
304:       { ρ[i][processor(j)] = visit_ratio[i][processor(j)] * (R + C);
305:         ρ[i][memory(j)] = visit_ratio[i][memory(j)] * L;
306:         ρ[i][inbound(j)] = visit_ratio[i][inbound(j)] * S;
307:         ρ[i][outbound(j)] = visit_ratio[i][outbound(j)] * S;
308:       }
309:     }
310: }

401: AMVA
402: { do {
403:   for all classes: class = 1 to P, initialize total_wait_time=0;
404:   { for each queueing node at all processing elements: 1 to P;
405:     { for each class: r = 1 to P, initialize sum = 0;
406:       { sum    += 1 + (n_i[r]-1)/n_i[r] q[r][node][old];   if (r == class)
407:                += q[r][node][old];                         for remaining classes r
408:                = 1.0;                                      for delay centers
409:       } q[class][node][new] = sum;                         total queue length at a node
410:     } waiting_time[class][node] = ρ[class][node] * q[class][node][new];
411:       total_wait_time[class]+ = waiting_time[class][node];
412:   } arrival_rate[class] = n_i[class] / total_wait_time[class];
413:
414:   Compute maximum_differences between the waiting time and arrival rate
415:       from current iteration with their values from the previous iteration;
416:   Assign      q[class][node][old] = waiting_time[class][node] * arrival_rate[class];
417:   } while ( maximum_differences > tolerance_level)
499: }
```

```
501: Performance_Measures
502: { Compute λ_net[class] using arrival_rate[class] and p_remote; /* Equation 5.10 */
503:   Compute processor_utilization using
504:       arrival_rate[class], runlength and context_switch_time; /* Equation 5.11 */
505:   Compute network_latency using
506:       waiting times at the switch nodes for each class; /* Equation 5.9 */
599: }

601: AMVA_SRP
602: { do {
603:   Identify secondary nodes required for the simultaneous possession, e.g. the node bus;
604:   Identify the set of primary nodes associated with each of the secondary nodes,
605:       e.g. {the memory, link_in and link_out} at a node associated with the node bus;
606:   for all classes: class = 1 to P, initialize sum2 = 0, total_wait_time = 0;
607:   { for each queueing node at all processing elements: 1 to P;
608:     { for each class: r = 1 to P, initialize sum = 0;
609:       { sum    += 1 + ((n_t[r]-1)/n_t[r]) q[r][node][old];   if (r == class)
610:                += q[r][node][old] (ρ[r][node]/ρ[class][node]);   for remaining classes r
611:                = 1.0;                                        for delay centers
612:       } q[class][node][new] = sum;                           total queue length at a node
613:     } waiting_time[class][node] = ρ[class][node] * q[class][node][new];
614:
615:     if (node is a primary node)          { Do not modify total_wait_time.}
616:     if (node is a secondary node)
617:     { sum2    = ((queueing delays at the secondary node and associated primary nodes)
618:                 *(total number of accesses through the secondary node))
619:                 + (ρ[class][node] at the secondary node and associated primary nodes);
620:       total_wait_time[class]+ = sum2; }
621:     if (node is neither primary nor secondary)
622:     { total_wait_time[class]+ = waiting_time[class][node]; }
623:   } arrival_rate[class] = n_t[class] / total_wait_time[class];
```

624: **Compute** maximum_differences between the waiting time and arrival rate
625: from current iteration with their values from the previous iteration;
626: **Assign** $q[class][node][old] = waiting\_time[class][node] * arrival\_rate[class]$;
627: } **while** (maximum_differences > tolerance_level)
699: }
