McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures
and Parallel Systems

Measurement and Modeling of EARTH-MANNA Multithreaded Architecture

Shashank S. Nemawarkar†
Guang R. Gao

ACAPS Technical Memo 96
July 24, 1995

†Department of Electrical Engineering
{shashank,gao}@macs.ee.mcgill.ca

(Preliminary Version)
Submitted for external publication.

ACAPS • School of Computer Science • 3480 University St. • Montréal • Canada • H3A 2A7
FTP: ftp-acaps.cs.mcgill.ca • WWW: http://www-acaps.cs.mcgill.ca/

Abstract

In this paper, we develop and apply an analytical model to predict the performance of McGill's EARTH-MANNA multithreaded multiprocessor system. The performance model is evolved from a closed queuing network model for multithreaded architectures reported in our earlier work [17]. In this work, we extend the original model to account for the complications due to realistic subsystem interactions and multithreaded workload in the EARTH-MANNA system. Our solution technique uses mean value analysis. We use simple heuristics to account for the subsystem interactions (simultaneous resource possession) and multithreaded workload. Inputs to the analytical model are architectural parameters derived from the EARTH-MANNA system measurements, and program workload parameters like number of threads, and their runlengths, supplied by the users. Outputs of the model include performance predictions such as processor utilization and network latency for remote accesses with split-phase transactions. We characterize the variation of these performance measures in terms of program workload and architecture characteristics.

Our analytical model has been validated through runtime measurements on the EARTH-MANNA system. A set of accurate software-controlled instrumentation tools has been developed to measure the network latency and processor utilization during actual multi-threaded program execution. The runtime measurements from EARTH-MANNA system on synthetic benchmarks match well with the analytical model (within 10% of the model predictions). The model predictions also conform with the measurements on real programs (in [16]). Our results demonstrate the tradeoffs of realistic costs of multithreading on the performance for fine-grain parallel program workload. The model is useful for both users (compilers) and system architects of multithreaded architectures and systems.

Key Words: Parallel and distributed systems, Advanced computer architectures, Multithreaded architectures, Performance models, Scalable systems, System and application program tuning.

Contents

List of Figures

List of Tables

1 Introduction

Multithreaded architectures have been proposed to tolerate the long latencies for communication and synchronization in parallel program executions, e.g. TERA [6], Alewife [4], *T [19], TAM [10], and EARTH [12]. On encountering a long latency operation in the execution of a task, a multithreaded processor rapidly switches context to another computation task. Thus the processor performance improves by maintaining a pool of ready threads for execution, as well as employing split-phase transactions for remote accesses and synchronizations such as the `GET_SYNC` and `DATA_SYNC` operations used in EARTH (see [9] for an in-depth introduction).

The main contributions of this work are as follows. We predict the performance of a multithreaded workload using realistic costs from multithreaded program executions. We validate the model predictions and characterize the performance of multithreaded architectures using system measurements. To our knowledge, this is the first work to address these issues for multithreaded systems.

We characterize the performance of McGill's EARTH architecture on multithreaded workload. Runtime measurements provide the performance behavior with real costs of multithreading operations. However, multithreaded program execution poses challenges to the performance measurements for the following reasons. First, a processor may be executing other threads after issuing a long latency operation, hence the measurement must accurately reflect the overlapped computation and communication. Second, the thread runlengths are of the same order as the network latencies, so overheads for the measurement can perturb the values of performance measures.

Similarly, performance predictions of multithreaded architecture using analytical models must account for the overlap of communication on one thread with the computation on another. Each thread is a sequence of computation followed by multithreading operations. Each communication may involve split-phase operations such as accessing the local memory, sending the messages on the network, receiving the responses, and performing some synchronization operations. Modeling the significant interactions of key components of a system under overlapped multithreaded computation and communication is the key to performance prediction. We develop an analytical model of EARTH-MANNA system, and validate the performance predictions using the system measurements.

The EARTH (an Efficient Architecture for Running Threads) architecture supports an efficient multithreaded execution model based on split-phase communication and synchronization operations [12]. Currently, the EARTH architecture is implemented on a 20-node EARTH-MANNA multiprocessor hardware testbed [16] and has been operational since the summer of 1994. Each EARTH node has an execution unit (Intel 860 XP) for computation, a synchronization unit for multithreading operations, a network interface unit (link) and a part of distributed memory. Resources on a node are accessible through a bus. EARTH nodes are connected across a high-bandwidth network, consisting of a hierarchy of crossbar switches.

In [17], we developed an analytical model based on closed queuing networks (CQN), to predict the performance of an abstract multithreaded multiprocessor system. The model takes into account the behavior of processors, memories and the interconnection network, and the interactions between them under various application program workloads. In this paper, we extend and apply our analytical model in [17] to predict the performance of the EARTH architecture. [1]

[1] WWW site `http://www-acaps.cs.mcgill.ca/{info/EARTH,~ shashank}` contains the papers [18, 17, 12,

Complex interactions among the resources in the EARTH system, and the characteristics of a realistic multithreaded workload are major challenges in extending the original analytical model [17]. First, on the architectural aspect, we model the interactions at each EARTH node in detail. For example, the bus at a node is a shared resource. Accesses to memory and network interfaces from processors and synchronization units, are arbitrated for the use of the bus. Each type of request, when granted, may hold the bus for a different amount of time before its release (called *simultaneous resource possession*). Second, for a program workload, the threads executing on different nodes (i.e. classes) may differ in thread characteristics, like the number of multithreading operations.

Our solution is to use mean value analysis (MVA) [20, 15]. We use simple heuristics to account for above extensions in the analytical model. First, our heuristic method combined with the iterative nature of approximate MVA allows us to formulate the simultaneous resource possession problem under one analytical model (whereas previous solutions use at least two models [13, 15]). The key observation is that every bus access experiences the contentions at all resources serviced by the bus i.e. the memory and network interfaces. Second, the MVA uses queuelengths at a functional unit for all classes to compute the waiting time for a new thread or access. Since the thread characteristics at different nodes may differ, we compute an accurate waiting time for a new access using the waiting time for the queuelength of a class at a functional unit.

Given program workload parameters, we use our performance model to derive performance measures of interest. Inputs to the analytical model are the architectural parameters derived from the EARTH-MANNA system measurements, and the program workload parameters like number of threads, their runlengths, and number of multithreading operations, supplied by the users. Outputs of the model include performance predictions such as processor utilization, network latency for remote access with split-phase transactions and synchronization. We also characterize the variation of these performance measures with program workload and architecture parameters such as the number of threads, the thread runlengths, and the number of multithreading operations.

We validate the model predictions using runtime measurements on EARTH system. For this purpose, we have developed an accurate software-controlled instrumentation tool to measure the performance parameters under these constraints. Our preliminary validation results are very encouraging. The runtime measurements from EARTH-MANNA system on synthetic benchmarks match within 10% of the analytical model predictions. These predictions also conform well with the measurements on the real programs (reported in [16]). Our results show that the EARTH system achieves up to 87% processor utilization using 4-8 threads with runlengths 1500 cycles. Lower runlengths yields a lower maximum processor utilization, but the speed up over single thread performance is high. Thus our results demonstrate the tradeoffs of realistic costs of multithreading on the performance for fine-grain parallel program workload. These results provide a strong evidence on the usefulness of our performance model to compilers and system architects to identify the performance bottlenecks in program and architecture in order to achieve best performance on a multithreaded systems such like EARTH.

Organization: The rest of the paper is organized as follows. In the next section, we outline the EARTH architecture testbed, the program workload for performance characterization, and the tools developed for runtime measurements. In Section 3, we present our analytical model,

16].

and discuss the assumptions of the model. We Section 4, we highlight the results of this paper. In Section 5, we characterize the performance of EARTH system on multithreaded workloads, and validate the analytical predictions using system measurements. In Section 6, we apply our performance model to make a performance tradeoff for workload distribution. In Section 7, we discuss related work. Finally in Section 8, we present concluding remarks.

2 Experimental Testbed

This section contains a description of the McGill EARTH-MANNA system relevant to this paper, the program workload to characterize the system, and the tools to gather the performance results from the system.

2.1 EARTH Architecture

The EARTH (an Efficient Architecture for Running Threads) architecture proposes efficient execution of the synchronization operations and the computations using different functional units [12]. Currently, the EARTH programming model is implemented on the MANNA multiprocessor, developed at GMD FIRST in Berlin, Germany [8]. The EARTH Threaded-C compiler offers a direct support for EARTH operations, by expanding them inline in order to reduce their overheads. A detailed description of the system appears in Appendix A.[2]

System: The EARTH-MANNA multiprocessor system consists of multiple high performance MANNA nodes across a high-bandwidth interconnection network. Figure 1 shows one MANNA node, and Figure 2 shows a multiprocessor configuration for EARTH-MANNA system. Each MANNA node emulates an EARTH node. Henceforth we refer to a node as an EARTH node. Each EARTH node consists of two Intel i860 XP RISC processors clocked at 50 MHz, 32 MB of DRAM, and a fast network interface called link. The network is based on 16 x 16 crossbar chips.

EARTH Node: Each EARTH node consists of an Execution Unit (EU), a Synchronization Unit (SU), and a part of distributed shared memory (see Figure 1). The EU and SU interact through *ready* and *event* queues maintained in memory at the same node.

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 fetches a *thread id* from the *ready queue*, and executes a thread to completion before moving to another thread. The EU places synchronization and remote accesses in the *event queue*. The SU fetches these requests from the *event queue*, in addition to the messages coming from remote processors through the network. The SU responds to requests, and also schedules threads for execution by placing their *thread id*s to the *ready queue*.

EU: Each EU executes a set of n_t threads (e.g., these threads may be n_t iterations of a `doall` loop as discussed later in Section 2.2). The number of instructions within a thread is the *runlength* of a thread, and its average is denoted by R. Number of read and write accesses to local memory is rw. After issuing a long latency memory access (whether local or remote), the EU context switches to another ready thread. C denotes the *context switch time*.

[2] In this paper, we will use the terms "EARTH", "EARTH-MANNA" and "MANNA" interchangeably to represent "EARTH-MANNA system". In reality, "EARTH" architecture is mapped on to the "MANNA" system.

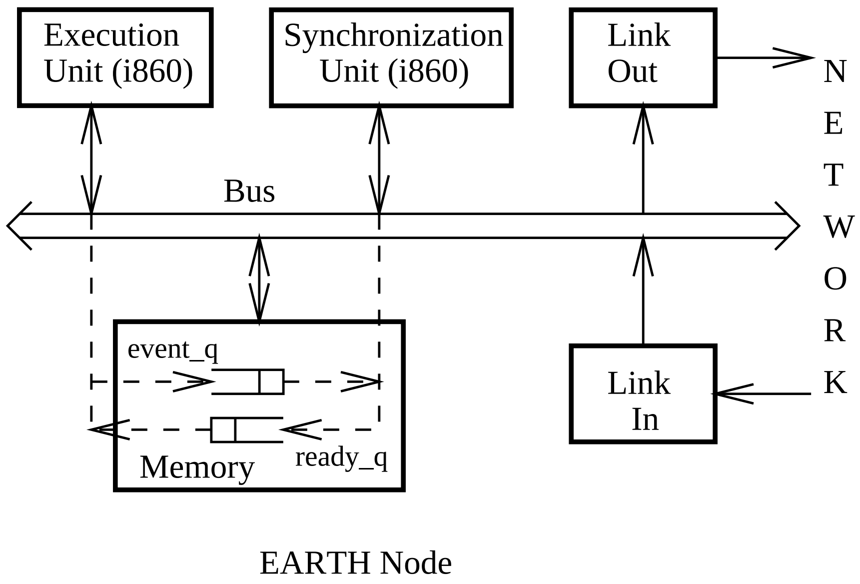

Figure 1: An EARTH-MANNA Node

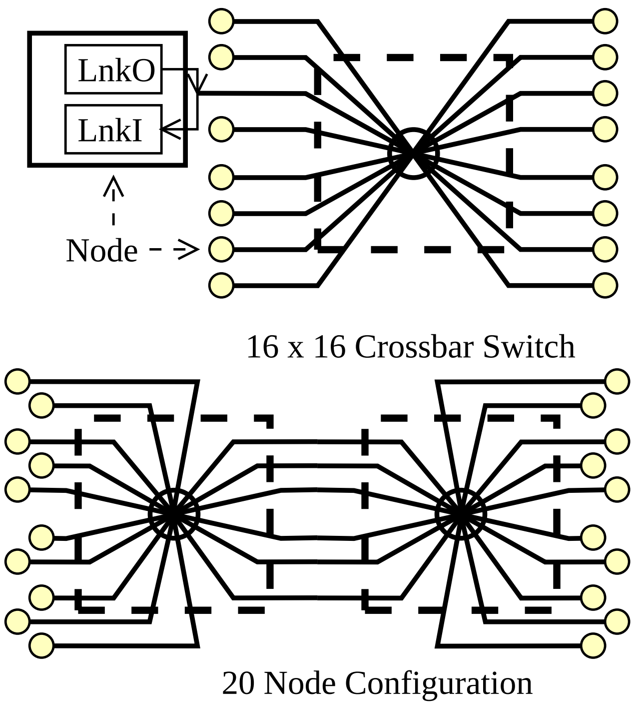

Figure 2: An EARTH-MANNA Multiprocessor System

SU: An SU reads the incoming messages from event queue (written by local EU), and the link_in nodes (network messages). The SU responds by performing a combination of: a read/write to local memory, a message send to remote SU, and a write to the ready queue (for local EU). Every processing of message at the SU takes SU_{serv} cycles, because the SU goes through a fixed, optimized set of events of each request.

Memory: When an EU or SU accesses the local memory, the request is serviced in 10 cycles (without queuing delay).

Link Interface: The network interface at an EARTH node consists of link nodes– *Link_in* and *Link_out*. Buffers at each link node can store up to 4 messages.

An SU sends a message to a remote SU through *Link_out* node. First, the SU checks for whether the buffer at Link_out node is not full. On a positive response, the SU sends out the message.

An incoming message from the network is buffered at the *Link_in* node. Local SU accesses the Link_in node to check for a message and its retrieval. Depending on the message type, more than one access to Link_in node may be necessary.

Bus: All functional units in an EARTH node communicate through a pipelined bus, e.g., an access to a link node by SU. A delay of *one* cycle is introduced on a bus access. The bus is held till the access to a functional unit is complete.

Crossbar: Each crossbar chip has 16 input channels and 16 output channels. A channel is 1 byte wide. Output channels operate in parallel. Each output channel selects an input channel for message transfer in a round-robin manner. It takes up to 32 cycles to transfer first byte to output channel. Once the connection is established, the data transfer is pipelined from link node to output channel.

The interconnection network is based on a hierarchy of crossbar chips. The processing nodes are connected to the leaves of the interconnection network (Figure 2 shows a 20-node configuration).

EARTH Threaded-C Language: The EARTH program model is implemented as an extension to the C language. These multithreading extensions provide a support for, the 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. For example, one remote datum is fetched using one `GET_SYNC` operation, and a remote store is performed using a `DATA_SYNC` operation. The `END_THREAD` operation performs a context switch. Explicit mention of these operations helps in obtaining the program workload parameters.

Refer to Appendix B for a synopsis of multithreading operations.

2.2 Program Workload

In this section we outline the program workload used to characterize the performance behavior of the EARTH-MANNA 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 interference at an EARTH node. The second workload characterizes the contentions at the node-network interface, and the interconnection network. We choose these workloads because the input parameters of interest are easy to adjust, and their effect on the system performance can be individually studied. (Appendix B provides details on the EARTH Threaded-C language.)

First workload characterizes the EU-SU interference on a bus at a node. A remote access reaches the node under investigation, through link interface and the SU. To process the access, the SU fetches the requested location from the memory, prepares a response, and send the response through link interface. For each bus access, the SU may experience a contention from the EU. The program workload ensures that the EU execution has three phases as shown in Figure 3. For *execute* phase, the EU does not contend on the bus. At the end of R cycles, the EU requests an access to its local memory. Shaded *read/write* phase indicates this access experiences a contention from the SU. The *idle* period indicates that bus access is with the EU, 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 3. We monitor the contention for the bus by varying the thread runlength.

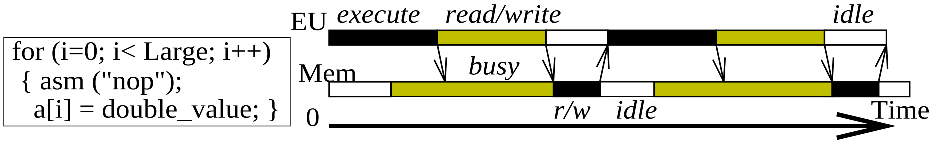

Figure 3: Workload for node characterization

Figure 4 shows our second workload for characterizing the performance of the network, and its interface to the node. Input parameters varied using this workload are: the number of threads, their thread runlengths, and the number of multithreading operations for each thread.

The program description is as follows. All EUs follows same execution behavior. Each EU fetches two arrays (`a` and `b`), computes their vector sum, and stores the result (in array `c`). For each thread, two `GET_SYNC` operations are initiated to fetch `j`-th elements. At a time, fetches for a block of elements ($C = high - low$) are initiated. Thus C is the number of threads executing in parallel. 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.

2.3 Measurement Tools

In this section we describe the tools used for this performance study. Tools for obtaining the performance measures on an executing system are invaluable to the task of performance tuning. The ease of use, the accuracy of measurement, and the perturbation to the system execution, are important aspects in the choice of tools.

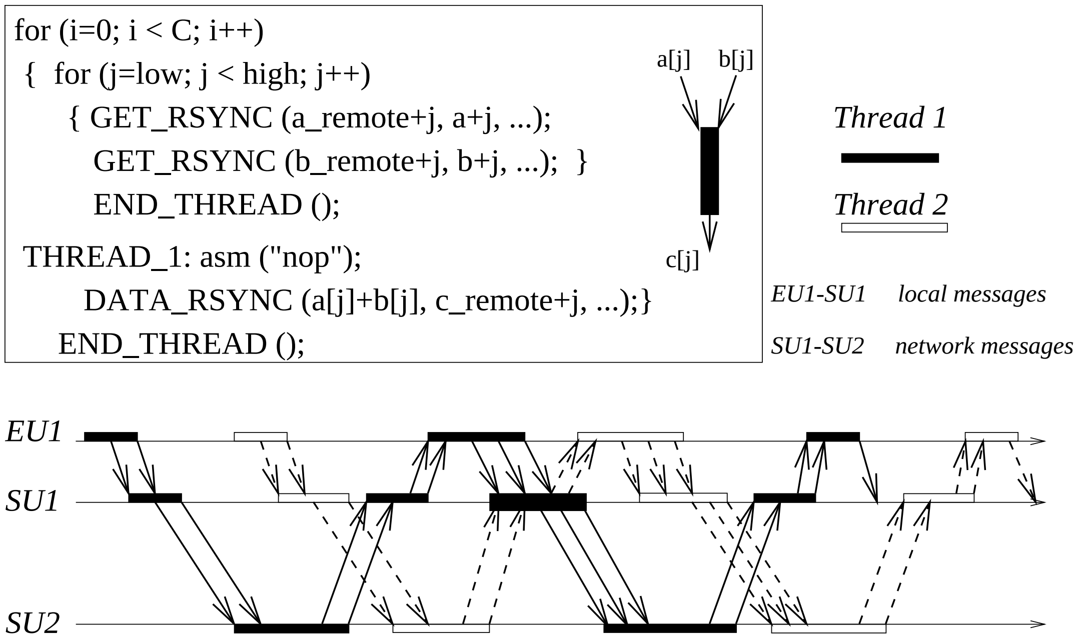

Figure 4: Workload with Multithreaded Operations

The key performance measures of interest are, processor utilization (i.e. at the EU), message rate to the network (i.e. SU-network interface), and the network latency (i.e. the latency for multithreading operations like GET_SYNC). Only the message rate to the network can be measured directly. Processor utilization is estimated using the idle time statistics. Network latency requires more elaborate measurements.

Network Latency

Network latency is the one of the most difficult performance measure to obtain from a multithreaded multiprocessor system. Multithreaded architectures overlap the communication on a thread with the computation on another. This mechanism prevents a precise measurement of the elapsed time between the initiation of a remote memory access, and the reception of its response. We consider following two choices for the measurement:

1. *Direct measurement*: Placing a time-stamp on a message is a simple way to measure the network latency. Three overheads involved in this process are: the maintenance of a timer, the increase in the message length to include time-stamp, and the computation of the network latency. These overheads perturb the measured value as well as the program execution, because thread runlengths and network latencies are almost of the same order of magnitude.

2. *Indirect measurement (Sampling)*: In this method, a thread is dedicated the task of measurement. It sends test messages to selected nodes, and measures the time till the receipt of their responses. This thread requires an exclusive access to an EU during the measurement. We refer to this node as the *dedicated* node. The latency measured by this method indicates how much contention a message suffered during one round-trip. Exclusive access to an EU for time measurement ensures that the dedicated node does

not delay the test message. So any delay, in excess of no-load latency value, is a result of the contentions at the network, and the remote node(s).

We have developed a tool to measure the network latency based on the second approach. The tool measures the network latency with good accuracy, and causes little perturbation to the program execution.

The functioning of this tool is as follows. The tool is a software instrumentation to the application program, i.e. a measurement thread is forked for execution at the dedicated node. Remaining nodes execute the application program. For each computation phase in the program, a measurement can be initiated at run time. *First*, the dedicated node sends a sample message *one at a time* to one of the nodes under investigation. *Second* a counter (maintained in register) measures the elapsed time, till the response from the remote node is received. *Finally*, if the measurement is to be continued, then the above process repeats by sending a new sample message. The tool reports the average latency for a `GET_SYNC` operation, and number of sample messages.

The accuracy of the tool is 5 cycles, i.e. 100 ns on 50 MHz Intel 860 XP. Execution time of the application program changes due to the measurement tool by less than 10%.

Processor Utilization

Utilization of the EU is obtained by measuring its idle time. Whenever there are no threads in *ready* queue to execute on, the EU measures the idle time. As soon as a thread arrives through *event queue* (i.e. from SU), the idle time measurement terminates. The accuracy of measurement is within one millisecond.

Message Rate to the Network

Message rate to the network is obtained by counting the number of remote accesses, and the duration of execution of a program block. For time measurement, the MANNA system provides a timer. The timer is updated every 400 ns. Each access to the times takes 2.5 microseconds.

When the statistics for processor utilization, and message rate are gathered, the execution time of the application program changes by less than 10%. So the effect on the values of processor utilization, and message rate is within 10%, even for small programs with execution time of 100 ms.

3 Analytical Model

In this section, we briefly outline our analytical performance model for the EARTH system. This model includes the extensions to our original model (in [17]) 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 on 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.

3.1 The Model and Its Assumptions

Our performance model of the EARTH system is based on closed queuing networks (CQN). Nodes in the CQN model represent the functional units in the EARTH system and edges represent their interaction through messages (see Figure 5). We use simple models of the subsystems and show their impact on the system performance.[3] We discuss the model in more detail, below. All symbols are summarized in Table 2, along with the experimental values for the system parameters.

We assume that the application program exhibits similar behavior at each node, and that the load is evenly distributed (like a Single-Program-Multiple-Data, SPMD, model) [11]. 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. The application program is a set of partially ordered threads. A thread is a sequence of instruction followed by a memory access. A thread repeatedly goes through the following sequence of states– *execution* at the processor, *suspension* after issuing long latency memory accesses, and *ready* for execution after the arrival of all responses. Threads interact through memory locations. Note that a long latency memory access is explicitly specified. This access goes via SU to local or remote memory (see Figure 1).

We now describe the assumptions in the analytical modeling of the EARTH system.

- All nodes in the performance model are single servers, with First Come First Served (FCFS) discipline.
- **EU or Processor**[4]: Each EU executes a set of n_t threads (say, n_t iterations of a `doall` loop are executed at each node). An EU is modeled as a *processor node*. The service time of a thread is exponentially distributed with a mean value of R cycles. During its execution, a thread performs rw local memory accesses, which are not long latency operations. For the context switch, a cost of C cycles is incurred. We assume that threads do not migrate, so threads at a node i belong to a class i in the CQN model.

  Processor utilization, is defined as the fraction of the time an EU performs useful work, and is denoted by U_p. Idling for a memory response or context switching is not considered as useful work.
- **SU**: A multithreading operation requires one or more visits to an SU. At each visit, the service time is exponentially distributed with an average of SU_{serv} cycles. We ignore the differences for local accesses and for bulk movement of data.

  λ_{net} denotes the rate of messages exchanged between the SU and the interconnection network during the program execution.
- **Memory Node**: A memory module is modeled as a *memory node*. The service time is exponentially distributed with L cycles as the average value.

  When an EU issues a long latency memory access, let p_{remote} be the probability of a remote access (i.e. $1-p_{remote}$ is the probability of local long latency access). For requests from a thread at processor i to memory at node j, $em_{i,j}$ denotes the *visit ratio*[5]. The

[3]The model for each subsystem can be improved to suit the requirements of a performance evaluation study.

[4]We use the notation "EU" and "processor" interchangeably.

[5]The *visit ratio* for a subsystem at a node j for a class i of threads (i.e. executing on processor at node i) in a closed queuing network is the number of times a thread belonging to the class i requests an access to that subsystem at the node j, between two consecutive executions of that thread at the processor. Note that the *visit ratio* can be a fraction.

value of $em_{i,j}$ depends on the distribution of remote memory accesses across the memory modules.

- **Link Nodes**: An SU sends a message to a remote SU through *Link_out* node. The service time at *Link_out* node is exponentially distributed with an average value of $lnkout_{serv}$ cycles.

  Similarly, Link_in node has a service time exponentially distributed with a mean value of $lnkin_{serv}$ cycles. Due to the high speed of operation of link nodes with respect to the processing time at SU, we assume an infinite buffer capacity at link nodes.

- **Bus**: The server at the bus has an exponentially distributed service time of bus_{serv} cycles. We assume that an access to a resource, has a delay of bus_{serv} (=1) cycle at the bus, and that the rest of the delay occurs at the resource.

- **Crossbar Switch Nodes**: Each input port of the crossbar switch node has exponentially distributed service time with mean value xin_{serv} cycles.

  Similarly, each output port has exponentially distributed service time with mean value $xout_{serv}$. 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, we assign the length of the message to the service time of input port, i.e. xin_{serv} (=8) cycles.

  The time taken by a message from its entry on the IN till its exit to the destination node is called the *network latency*.

We make the following assumptions in our CQN model to simplify its analysis: (1) Every thread in a processor that is waiting for a memory request will eventually get its request in finite time. So there is no deadlocking of threads waiting for memory requests due to finite resources. (2) There are only finite number of active threads in a processor at any instant. Thus there is no deadlocking due to finite buffer capacities of real network switches. These two assumptions are reasonable for practical systems like EARTH [12], TERA [6] and Alewife [4].

3.2 Solution Technique

Our solution technique uses approximate mean value analysis (AMVA) [15]. Two advantages of our approach are the computational efficiency of AMVA, and the amenability of AMVA to heuristics. We propose two simple heuristics to predict the performance of the CQN model shown in Figure 5. The first heuristic accounts for the multiple classes in the CQN, and the second heuristic accounts for the thread accesses exhibiting *simultaneous resource possession* property.

An essential idea of AMVA is that the queuelength seen by an arriving access at a node is equal to the time averaged queuelength there with one less customer in the network [15]. For a CQN with multiple classes the same idea holds, and that the queuelength for a system with one less thread (access) in the class of the arriving access is considered. However, we note that AMVA ignores the effect of differences in service demands for different class of accesses, on the waiting time. Our first heuristic takes into account this difference in service time.

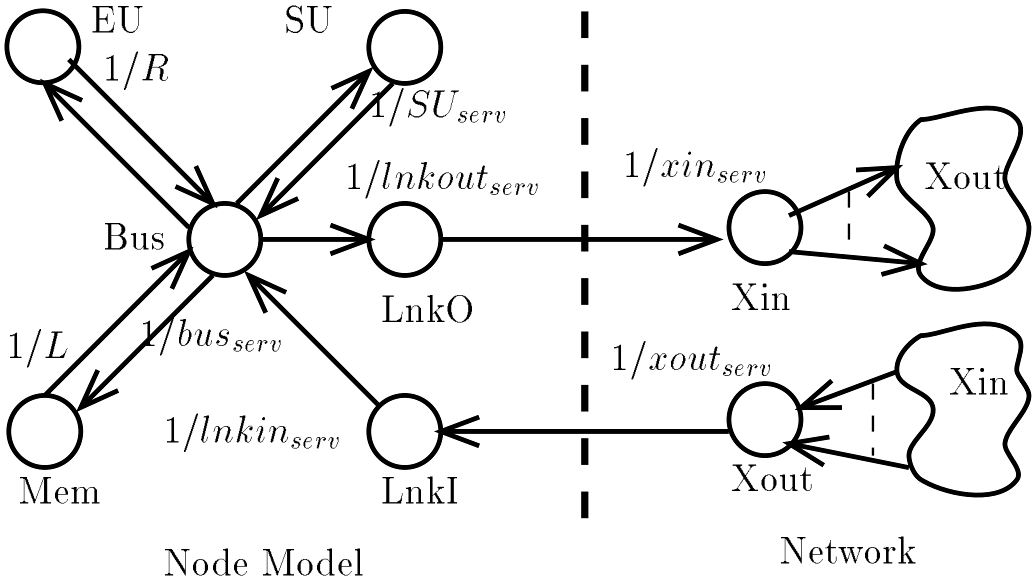

Figure 5: Queuing Network Model

The CQN model shown in Figure 5, does not have a *product-form* solution for following reasons:

1. An access to one of the resources, like memory, prevents the use of bus for accessing other resources (see Baskett *et al.* [7] and Reiser [20]).
2. Pipelined transfers at the crossbar switches.

We show that a simple heuristic provides sufficiently accurate performance prediction of EARTH system, without the need for solving multiple analytical models (like in [13]).

The core AMVA algorithm is outlined in Appendix C (adapted from [15]). With n_t threads on each processor in the system, for each class i of threads AMVA evaluates:
(i) the arrival rate λ_i of threads to (and from) processor i; (ii) the waiting time $w^*_{i,m}$ at each node m; and (iii) the queue length $n^*_{i,m}$ at each node m.
Now we discuss the heuristics to this AMVA algorithm, for performance prediction of EARTH system.

Heuristic for Multiple Classes:

We briefly outline the heuristic to account for different service demands for accesses from different classes (processors). Let us consider the waiting time for a newly arrived access from class i. In MVA (see Figure 10 in Appendix C), Step 2(a) provides the queuelength seen by this access. However the waiting time for this access, given by step 2(b) is inaccurate, because accesses from different classes may have different service times. So an accurate composition of the waiting time at node m is given by Equation 1. (Refer to Table 2 for notations.)

$$w^*_{i,m}(N) \;\; = \rho_{i,m} \left[1 + \left(\tfrac{N_i - 1}{N_i} n^*_{i,m}(N) \right) \;\; + \sum_{j=1, j \neq i}^{P} \frac{\rho_{j,m}}{\rho_{i,m}} n^*_{j,m}(N) \right] \tag{1}$$

Heuristics for the Node Model:

Now we show how to incorporate the effect of *simultaneous resource possession.* Consider a local memory request issued by an EU (see Figure 3). This request contends with other memory accesses (say, a memory access by the SU is already in progress). So a memory access does not get the bus, till the previously issued requests are serviced. Similar wait period is encountered for the memory accesses by the EU, if the SU is accessing a Link node. In effect, each bus request observes the waiting queue at all of the resources. Although each resource is modeled as a separate node (i.e. a server and associated queue), we model the waiting time at the bus as the sum of the waiting time at all the resources— Link nodes, and the memory. Note that the service time for each bus request is that of the resource being accessed. Equation 2 provides the total wait time for a class i of threads at a node m. This value is used to compute the throughput for class i in Step 3 (of Figure 10).

$$\begin{aligned} w^*_{i,m,bus\ access} = \quad & (em_{i,m} + e_{i,m,bus} + e_{i,m,lnko} + e_{i,m,lnki}) \\ & \times[(w^*_{i,m,mem} + w^*_{i,m,bus} + w^*_{i,m,lnko} + w^*_{i,m,lnki}) \\ & -(\rho_{i,m,mem} + \rho_{i,m,bus} + \rho_{i,m,lnko} + \rho_{i,m,lnki})] \\ & +(\rho_{i,m,mem} + \rho_{i,m,bus} + \rho_{i,m,lnko} + \rho_{i,m,lnki}) \end{aligned} \quad (2)$$

Another important modeling consideration is for the access times for the resources, and their impact on the thread runlength at the EU. Consider the the execution of a thread on the EU. There are two distinct phases, one is the execution on the data in the cache, and second is the memory access (see Figure 3). During the first phase, the cache at the EU contains the necessary data for the computation. So the EU does not participate in the bus contentions. In the second phase, the EU accesses the memory, and goes through the waiting time to acquire the bus. Till the EU receives the response, and proceeds with the computation, the EU idles. However, the EU is not free to perform any other operation.[6] We model the duration of the second phase as the waiting time at the system resources. This duration is modeled as the idle phase for the EU. We believe that this phase does not significantly affect the performance of the EU. Also note that there is no mechanism (in software) which can measure this idle time incurred at the EU. In the case of SU, we have included the effect of bus accesses on the SU processing time (using Equation 2 and Step 4 in Figure 10).

Heuristics for the Node-Network Interface:

Three mechanisms at the node-network interface facilitate the data transfer from one node to another. First is a pipelined transfer from the buffers at the Link_out node to an input channel of a crossbar switch. Second is its inverse, i.e. a pipelined transfer from the output channel of a crossbar switch to the Link_in node. Third is a round-robin selection of the input channels at each output channel, in a crossbar switch. Note that the data transfer mechanism between the crossbar switches on the interconnection network is same as the first two mechanisms.

For the first two mechanisms, the buffer spaces at Link nodes, as well as the crossbar switch is limited. However, the service time at each of these units is much lower than that of the SU, which sends each message. So we assume that the buffer space is infinite (i.e. it has no impact on performance). We have verified our assumptions, and will report these queue lengths in the final version.

[6] 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.

Third mechanism is intended to provide a fair selection of all input channels. So for two successive requests from one node with same destination, a time lag of 32 cycles (the selection time for 16 channels) is introduced. We model this time lag as the service time of the output port ($xout_{serv}$).

Performance Measures :
Using AMVA and the above mentioned heuristics, we compute the arrival rate λ_i of threads to (and from) processor i, the waiting time $w^*_{i,m}$ at each node m, and the queue length $n^*_{i,m}$ at each node m.

Based on λ_i, $w_{i,m}$, service times and visit ratios, we obtain the following performance measures.

1. *Observed Network Latency* : The average network latency S_{obs} includes the queuing delay and service time at each crossbar switch, and at the link interface of each node. So S_{obs} for a processor at node i, 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 the switches in the IN:

$$\begin{aligned} S_{obs} \quad &= \textstyle\sum_{j=1}^{P} \Big((w^*_{i,j,lnki} \times e_{i,j,lnki} + w^*_{i,j,lnko} \times e_{i,j,lnko}) \\ &\quad + (w^*_{i,j,xin} \times e_{i,j,xin} + w^*_{i,j,xout} \times e_{i,j,xout}) \Big) \end{aligned} \tag{3}$$

   where $w^*_{i,j,k}$ is the waiting time at functional unit k of the node j, for request from node i.

2. *Message Rate to the Network*: λ_{net} is the average message arrival rate from a processor i to the IN. λ_i is the arrival rate of threads, and a fraction p_{remote} is sent to the IN. When each thread issues x `GET_SYNC` messages, $\lambda_{net,get-sync}$ is

$$\lambda_{net,get-sync} \quad = \lambda_i \times p_{remote} \times x \quad = \frac{n_t \times p_{remote} \times x}{wait\ time\ for\ a\ class\ at\ each\ node} \tag{4}$$

3. *Processor Utilization*: U_p is obtained from the arrival rate of threads λ_i at processor i and their runlength R:

$$U_p \quad = \quad \lambda_i \times R \tag{5}$$

In Section 5, we validate some of our analytical results by comparing with program executions on the EARTH-MANNA system. We use the above performance model to analyze the performance of processor and IN subsystems in Section 6.

4 Results

In this section, we present the highlights of the results on the model validation, and performance optimizations, when using our model for performance prediction of the EARTH system. To illustrate our results, without loss of generality, we analyze the model described in Section 3 as a case study with the default parameters in Table 2. Values of the architectural parameters are obtained experimentally, in the absence of contention. Each experiment mentions the values of the workload parameters.

The highlights of the results are as follows.

- We show that, for a given architecture and a given program workload, our analytical model permits a user to derive the network latency for a remote access operation of interest (in this case, GET_SYNC). We show how the accesses from the EU and the SU contend for the resources at a node, and increase the latency of a remote operation – such as the split-phase GET_SYNC operation: a frequently used and representative operation in EARTH programs. Performance measurements from the EARTH system matches to within 5% of the latency predictions by our model. The key success of the model is the heuristic proposed in Section 3.2 about how contentions on the bus increase during an access to a functional unit, because of a simultaneous possession of the bus. The model also characterizes the behavior of the crossbar switch on the latency, in the EARTH system. Our model provides a characterization of the network latency variations with the thread runlength, and the number of threads at each EARTH node. Our results demonstrate that our model indeed accurately reflects resource contentions and class dependent service time variations under the overlapped multithreaded computation and communication.

- We show that, for a given architecture and a given program workload, our analytical model allows us to derive a key performance measure of interest— the processor utilization, U_p. We show the characterization of the processor utilization with the thread runlengths, and the number of threads. The measurement results on EARTH match very well with the model predictions of U_p. With 4-8 threads of runlengths more than 1500 cycles, processor utilizations above 85% are achieved.

- We illustrate the usefulness of our model to compiler writers for performance optimizations on a program workload, in the presence of long latency operations. We show, for the EARTH architecture, how to choose thread runlengths and the number of threads, such that the processor utilization is high. We also show the effect on the latency of remote operations for these choices of workload parameter values. The measurements conform that these choices indeed yield a very high U_p ($> 85\%$).

- Measurements of latencies and processor utilizations on real programs (reported by Maquelin et al [16]) confirm with our predictions and measurements.

We present results as follows. In Section 5, we characterize the latency for GET_SYNC operation, due to contentions at a node and on the network. We investigate the effect of workload parameters on the network latency and the processor utilization. In Section 6, we show an example of optimizing a program workload in presence of remote access operations.

5 Validation of Performance Model

In this section, we show how to derive performance measures– the latency for a GET_SYNC operation and the processor utilization. We characterize the effect of contentions at a node and the network on these performance measures. We report the measurements from the EARTH system on synthetic benchmarks. These performance results are used to validate the model predictions.

5.1 EU-SU Interaction at a Node

Latency for GET_SYNC operation (also called as GET_SYNC latency, $L_{get-sync}$): The time between initiating a GET_SYNC operation from an EU and receiving its response at the EU, is the GET_SYNC latency. $L_{get-sync}$ provides a measure of waiting time due to contentions at various functional units. Its runtime measurement has a good accuracy, even when the system is heavily loaded (see Section 2.3). So an accurate characterization of the system performance is possible.

The following sequence of operations comprises a GET_SYNC operation. (The numbers in brackets indicate an estimated number of bus accesses.) The EU at the dedicated node requests the local SU for a GET_SYNC operation (2). The SU reads the request from memory (2), prepares a message to a remote SU, and launches it on the network (3). The remote SU receives the request through Link_In node (2), reads the requested memory location (1), and sends the response through Link_Out node (3). At the dedicated node, the SU receives the response through its Link_In node (2), writes the value in local memory (1), and sends a completion signal to the EU (2).

The GET_SYNC latency is the sum of waiting time for a GET_SYNC access each functional unit. Let us consider the delay at a node. An access through the bus observes the contentions at all node resources (memory and link nodes). Using the sequence of accesses for GET_SYNC operation described above, and the heuristic for simultaneous resource possession (Section 3), we compute the GET_SYNC latency:

$$\begin{aligned} L_{get-sync} \quad &= (w_{P,P,EU-getsync} + 12 \times w_{P,P,bus} + 3 \times w_{P,P,mem}) \\ &+ \frac{1}{p_{remote}} \times (4 \times w_{P,P,SU-getsync} + w_{P,P,lnko} + w_{P,P,lnki}) \\ &\quad + \frac{1}{p_{remote}} \times \Big[\textstyle\sum_{j=1}^{P-1} 2 \times w_{P,j,SU} \\ &\quad + (w_{P,j,lnko} + w_{P,j,lnki} + w_{P,j,mem} + w_{P,j,bus}) \times 6 \\ &\quad - (\rho_{P,j,lnko} + \rho_{P,j,lnki} + \rho_{P,j,mem} + \rho_{P,j,bus}) \times 5 \Big] \end{aligned} \tag{6}$$

In the above equation, P is the index of the dedicated node for measurement. $w_{i,j,k}$ indicates the waiting time at the functional unit k of the node j, for the GET_SYNC operation from node i. $\rho_{i,j,k}$ indicates the service demand for each access to k. Since the measurement thread sends one GET_SYNC message for each runlength, $L_{get-sync}$ also equals $\left(\frac{1}{\lambda_P} - R_{measurement\ thread} - 2\rho_{P,P,bus} - 2\rho_{P,P,mem}\right)$.

Our workload for characterizing the node behavior is shown in Figure 3 (see Section 2.2). At the node under investigation, the EU access its local memory every R cycles. The SU contends for the system bus only when the test message for latency measurement arrives. With a variation in thread runlength, we monitor the contention on the bus.

Figure 6 shows the effect of thread runlength on the GET_SYNC latency (in number of cycles). Values are obtained from the analytical results and the measurements from two 2-node EARTH-MANNA systems (one with a crossbar, and other without a crossbar). Closely spaced symbols show the experimental values 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 3,000–10,000 samples.

The difference between analytical results and the measurements from the system with crossbar is less than 5%. In the second system, the absence of crossbar decreases the latency value by 25-35 cycles. From Figure 6, we note the following:

- A decrease in the runlength for local accesses, significantly increases the contention. A rapid increase in the latency below $R = 10$ is observed, because the memory is almost continuously servicing 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 remote node makes 6 bus accesses to respond to a `GET_SYNC` message. A lower number of accesses will decrease the latency for an operation. Thus the incurred latency for a `GET_SYNC` operation is strongly governed by the implementation of `GET_SYNC` operation.

Maquelin et al. [16] report a no-load latency of 355 cycles (including 39 cycles overhead at the EU). These values conform with the model prediction and the measurement in Figure 6.

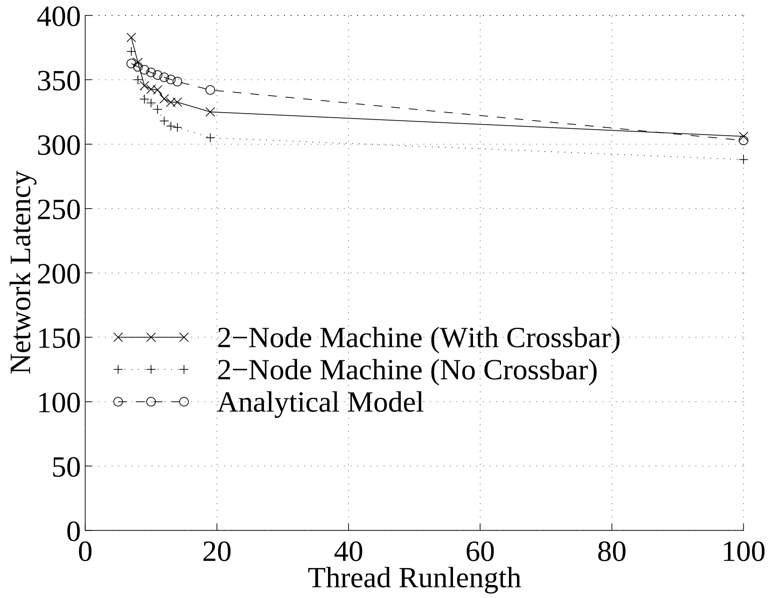

Figure 6: GET_SYNC Latency Measurement for EU-SU Contention

5.2 GET_SYNC Latency on Unloaded Network

Now we study the effect of delay at a crossbar switch on the `GET_SYNC` latency. Figure 7 shows the `GET_SYNC` latency (in cycles) with respect to number of nodes accessed by the dedicated node. In Figure 7, *Same Crossbar* means that the dedicated node is on same crossbar as the nodes under investigation. *Remote Crossbar* means that the dedicated node is on a different crossbar (see 20-node configuration in Figure 2).

The program workload ensure that no messages other than `GET_SYNC` samples traverse the network. So, ideally the delay at a crossbar switch should not change, irrespective of the destination pattern for sample messages. The reason for the `GET_SYNC` latency variation in Figure 7 is a round-robin selection procedure for an input channel. The performance model assumes a stepped increase up to 4 nodes. Beyond 4 nodes, the models assumes a small linear increase in the delay at the crossbar. Figure 7 shows that the model predictions are within 5% of the experimental values for two configurations.

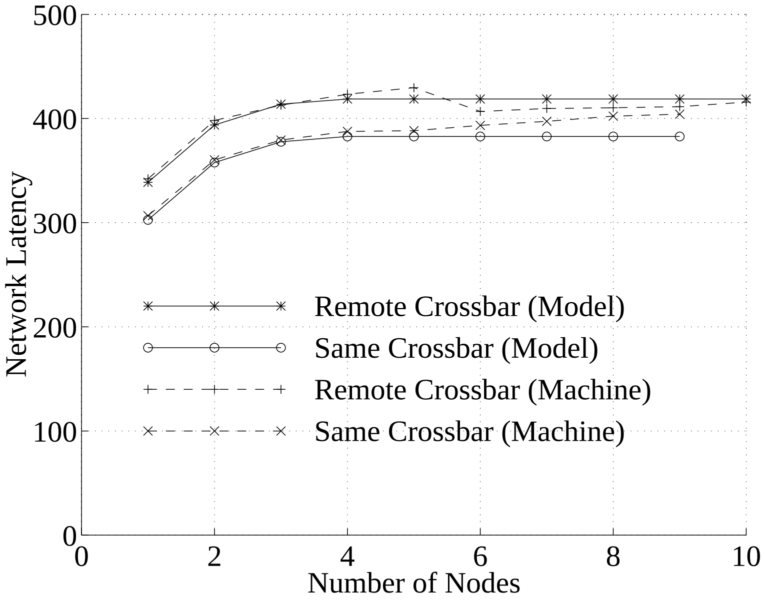

Figure 7: Effect of Crossbar Switch on GET_SYNC Latency

5.3 Effect of Workload Parameters on GET_SYNC Latency

Let us consider the multithreaded workload discussed in Section 2.2 (see Figure 4). The program workload consists of n_t (= 1 to 16) threads at each node. Thread runlengths are R (=1500, 3000) cycles. Each thread issues three multithreading operations. The workload parameters on the dedicated node are: $n_t = 1$, $R = 200$ cycles, and one `GET_SYNC` operation.

The effect of increasing the number of threads on `GET_SYNC` latency is shown in Figure 8. The number of nodes (including the dedicated node) is 8. The variations in `GET_SYNC` latency with thread runlengths and number of threads, are similar for the model predictions as well as the measurements on EARTH system. The model predictions are within 5% of the measurements. For $n_t = 16$, the difference is 20%. A probable cause for the discrepancy is that accurate values of architectural parameters cannot be measured directly. From Figure 8, we note that:

1. With increased runlength, a thread issues a remote operation at a longer interval (i.e. low message rate). A lower message rate reduces the contention at resources. Hence the `GET_SYNC` latency decreases.

2. Increase in the number of threads increases the number of contending accesses in the system. So the `GET_SYNC` latency increases.

3. At high runlengths, $L_{get-sync}$ saturates beyond $n_t = 8$. For low thread runlengths, $L_{get-sync}$ increases rapidly with n_t. (We will see experimental values of $L_{get-sync}$ for $R < 1500$ in Section 6.)

GET_SYNC latency values also show the significance of heuristics on multithreaded workload (in Section 3.2). For original MVA, we did not observe almost any change (< 1 cycle) in $L_{get-sync}$ values, even when the number of threads are increased from 1 to 16, and thread runlengths changed by an order of magnitude. The reason is that even for $n_t = 16$, the queuelengths at various functional units are very small (< 0.1) in comparison to those at the execution units. The use of heuristics on multithreaded workload illustrates in Figure 8 that $L_{get-sync}$ values vary significantly with the number of threads as well as thread runlengths.

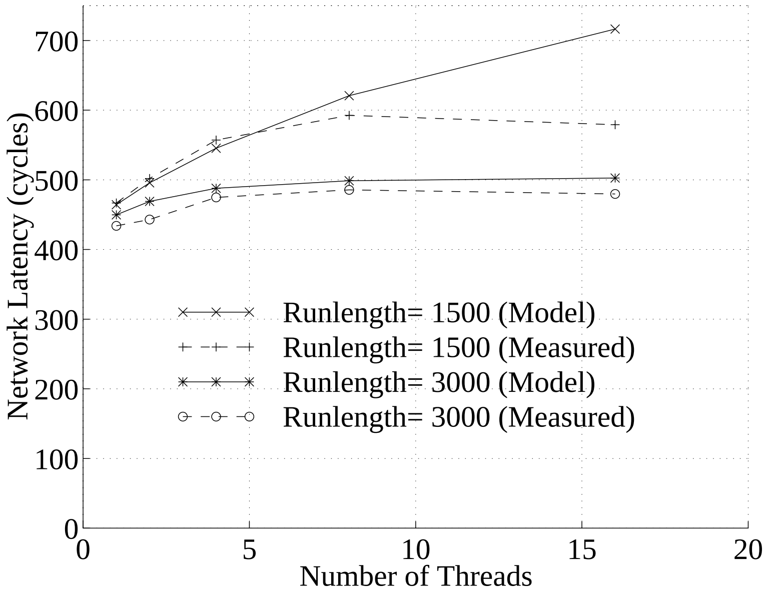

Figure 8: Effect of Number of Threads and their Runlengths on GET_SYNC Latency

5.4 Processor Utilization

In this section, we derive a key performance measure– processor utilization. Then we discuss the effect of a multithreaded workload on U_p.

Given the input workload parameters such as the number of threads and their runlengths, we solve the performance model using AMVA, for architectural parameters in Table 2. For a thread executing on a processor i, AMVA yields the arrival rate λ_i at the processor i. A thread keeps the processor busy, on average, for R cycles. So the processor utilization is $\lambda_i \times R$. A change in input parameters may affect λ_i, thereby changing the value of U_p.

We again consider the multithreaded workload for which Figure 8 shows the GET_SYNC latency values. Figure 9 shows how the number of threads and their runlengths affect the processor utilization (in percentage). We observe that the model predictions match within 5% of system measurements. At $n_t = 1$, the discrepancy is up to 20%. A probable cause for this discrepancy is the modeling assumption that a processor idles when local memory is accessed. Since this idle time cannot be measured, our measurements assume that the processor is busy under this situation. We observe from Figure 9 that:

1. A higher thread runlength yields a higher U_p value.

2. U_p values increase with n_t. This is an essence of 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 higher speedup occurs with n_t, if R is low. However the absolute U_p values are lower at low runlengths.

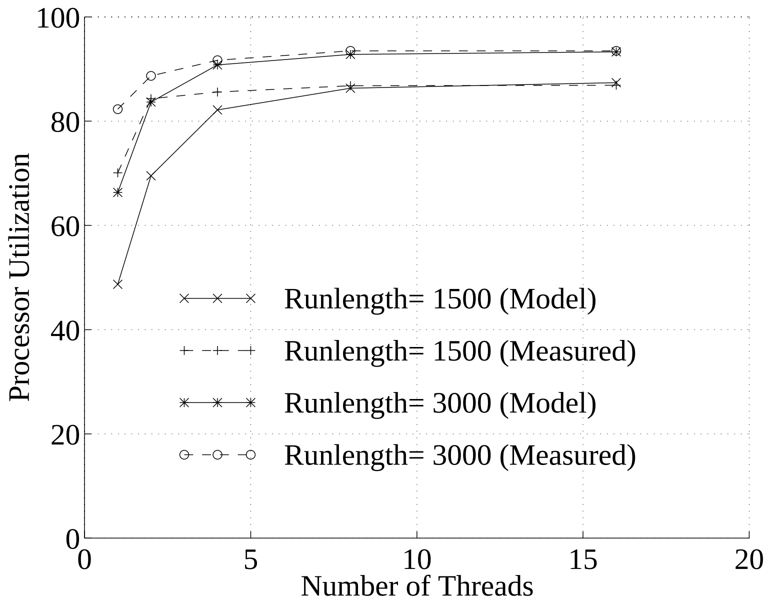

Figure 9: Effect of Number of Threads and their Runlengths on Processor Utilization

6 An Example of Workload Optimization

Now we show a specific example of how a compiler can use the performance model to optimize the program workload characteristics to achieve high processor utilization.

Table 1 shows analytical predictions of processor utilization and network latency for various thread runlengths and number of threads. System measurements for corresponding parameters are also shown. The system configuration uses 8 nodes. $L_{get-sync,ded}$ indicates the latency for a `GET_SYNC` operation issued by the dedicated node (for runtime measurements). No-load value of $L_{get-sync,ded}$ for an 8-node system is 402 cycles. U_p values are for the nodes executing the application program. "SU" values indicate the number of cycles to process a message at a remote synchronization unit. At no-load 60 cycles are required.

Let us consider the rows marked a in Table 1, and the values which are in **bold face**. When we decrease the input parameter R, we use a larger number of threads. We note that increasing n_t from 1 to 8 results in a 57% increase in analytical $L_{get-sync,ded}$ values, thus the contentions in the system increases. Analytical U_p values reach a maximum for $n_t = 2$ and 4. However at large n_t, U_p values are low. The system measurements confirm these predictions. These results indicate that high processor utilization is obtained when:

- number of threads is moderately high i.e. from 2 to 8.
- thread runlength is high.

On granularity of threads, our results indicate that application with thread runlengths higher than 1500 cycles yield higher values of U_p. However for lower runlengths, contentions at various functional units are significant (see "SU" values from Table 1). So network latencies increase rapidly, and U_p values are low. Most applications reported by Maquelin *et al.* [16] have thread runlengths of the order of 10,000 cycles to 25,000,000 cycles. The N-Queens program (in [16]) has the smallest runlengths of 700 cycles. U_p value for one-node system is 88%, and 8-node system yields a speedup of 6.4 (i.e. U_p = 70%). These values are similar to the results in Table 1. In summary, our results show that EARTH system can efficiently support applications with granularity up to 1500 cycles.

	Input		Analytical Predictions			Measurements	
	R	n_t	SU	U_p	$L_{get-sync,ded}$	$L_{get-sync,ded}$	U_p
a	6000	1	**72.1**	**74.3**	**433.3**	**416.9**	**91.8**
		2	74.9	87.7	440.2	421.4	93.9
		4	76.4	93.9	443.8	434.1	95.4
	3000	1	78.8	66.3	449.7	433.9	82.3
a		2	**86.8**	**83.7**	**487.8**	**443.0**	**88.7**
		4	94.8	90.8	498.7	474.8	91.7
		8	99.5	92.8	502.6	485.5	93.5
	1500	2	98.2	69.5	495.6	501.9	84.3
a		4	**120.2**	**82.2**	**545.3**	**557.0**	**85.6**
		8	155.2	86.3	620.7	592.5	86.8
		16	201.3	87.3	716.5	579.1	86.9
	750	4	132.1	32.2	571.2	787.2	65.5
a		8	**182.7**	**39.3**	**678.0**	**1016**	**72.3**
		16	279.6	45.6	874.6	1035	72.3

Table 1: An Example of Workload Optimization

7 Related Work

Performance evaluation studies on multithreaded architectures can be classified as: analytical models [21, 5, 3, 14, 18, 1, 25] and trace-driven simulations [24, 22].

Queuing Network and Petri Net Models: Petri Net models of multithreaded processors are proposed by Saavedra et al [21], and Alkalaj and Bopanna [5]. They use state-space analyses to derive the system performance, so modeling a large system and the contentions at subsystems, is computationally expensive. Yamamoto et al [25] propose an analytical model for a superscalar multithreaded processor. Their model accounts for the data and structural hazards during the instruction execution. Nemawarkar et al [18], and Adve and Vernon [1] proposed closed queuing network models using mean value analysis. These closed system models capture the system behavior more realistically, and are useful to analyze large multiprocessor systems. Analytical models in [5, 18, 1] are validated through the simulations of petri net and queuing network models. Saavedra et al [21] report a validation using simulation results (from [24]).

Other Models: Agarwal [3] proposed an analytical model based on cache parameters. Johnson [14] extended this model to include the feedback effect of network on the performance. Both models are validated using simulations of Alewife system [4].

Simulations: Weber and Gupta [24] performed trace-driven simulations with strategies for switching contexts, constant context switching times, and constant shared bus latencies. Their considered a workload with multiple copies of a program trace as multiple threads. Thekkath and Eggers [22] extended a similar approach using an analytical model [2] for the network performance. Waldspurger and Weihl [23] 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 paper, 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 multiprocessor system. This is a significant extension over previous studies. By addressing above issues, we believe that our work is a first strong evidence on the usefulness of the analytical models for performance optimizations on multithreaded systems.

8 Conclusions

In this paper, we developed and applied an analytical model to predict the performance of the McGill EARTH-MANNA multithreaded multiprocessor system. We characterized the performance using realistic costs of multithreaded operations. We validated the performance predictions using measurements on the EARTH system. To our knowledge, this is the first work to provide the performance characterization and performance prediction for multithreaded workload using system measurements.

Our performance model is based on closed queuing networks. Our solution technique uses approximate mean value analysis (AMVA). We used simple heuristics to account for realistic subsystem interactions (like simultaneous resource possession) and the multithreaded workload.

We showed how, given workload parameters, to derive performance measures, like processor utilization and network latency for the EARTH system. We characterized these performance measures with program workload and architecture parameters.

We developed an accurate software-controlled tool for the run-time measurements on the EARTH system. The runtime measurements on the EARTH system for synthetic benchmarks conform to within 10% of the model predictions. The model predictions also conform with the measurements on large programs (reported in [16]). Our results demonstrate the tradeoffs of realistic costs of multithreading on the performance for fine-grain parallel program workload.

Our results provide a strong evidence on the usefulness of the performance models to compilers and system architects for performance optimization on multithreaded systems.

9 Acknowledgments

This work was supported by MICRONET, Network Centers of Excellence, Canada. We would like to thank the EARTH group at McGill University, and the MANNA group at GMD, Germany, for building the EARTH-MANNA system, and making it available for experimentation.

References

[1] 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.

[2] A. Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[3] A. Agarwal. Performace tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[4] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In *Proc. of the 22nd Int'l. Symp. on Computer Architecture*, pages 2–13, 1995.

[5] 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. IEEE.

[6] 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.

[7] 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.

[8] 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.

[9] J.B. Dennis and G.R. Gao. Evolution of multithreaded architectures. In Robert A. Iannucci, Guang R. Gao, Robert H. Halstead, Jr., and Burton Smith, editors, *Multithreaded Computer Architecture: A Summary of the State of the Art*, chapter 1. Kluwer, Norwell, MA, 1994.

[10] David E. Culler et al. 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.

[11] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, Dec. 1986.

[12] 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.

[13] P.A. Jacobson and E.D. Lazowska. Analyzing queueing networks with simultaneous resource possession. *Communications of the ACM*, 25(2):142–151, February 1982.

[14] 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.

[15] 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.

[16] Olivier Maquelin, Herbert H. J. Hum, and Guang R. Gao. Costs and benefits of multithreading with off-the-shelf RISC processors. In *Procs. EURO-PAR'95*. Springer-Verlag, 1995. To appear.

[17] 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. Submitted for journal publication.

[18] 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.

[19] 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.

[20] M. Reiser and S. Lavenberg. Mean value analysis of closed multichain queueing networks. *Journal of ACM*, 27(2):313–322, April 1980.

[21] 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. ACM.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

A McGill EARTH-MANNA System

The EARTH (an Efficient Architecture for Running Threads) architecture proposes efficient execution of the synchronization operations and the computations using different functional units [12]. 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 1). 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 1). 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 2). 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 B provides the details.

B 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()).

C Approximate Mean Value Analysis

We assume the same terminology and notation as described earlier in Section 2. For n_t threads on each of the P processors in the system, the AMVA[7] computes (see Figure 10):
(i) the arrival rate λ_i for the threads belonging to each processor i; (ii) the waiting time $w^*_{i,m}$ at each node m; and (iii) the queue length $n^*_{i,m}$ for population vectors $N = (n_t, \ldots, n_t)$ and $N - 1_i$. Population vector $N - 1_i$ indicates that there are $n_t - 1$ threads at processor i, and other processors have n_t threads each. Statistics for population vector $N - 1_i$ 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 N indicate the steady state performance measures when n_t threads are present at each processor.

AMVA makes an initial guess for queuelengths at each node, for P classes of threads, assuming that the thread population in the system is N. From Steps 2 to 5, the algorithm iteratively refines this guess till the maximum difference between queuelengths obtained from two successive iterations is within a *tolerance* value. Step 2 computes the queuelength $n^*_{i,m}$ at each node m, for a class i when the population is $N - 1_i$. The *approximation* function used here, adds the contribution, to the queuelength for a class i, from a class j according to the number of threads in class j. So only when $i = j$, a scaling of $\frac{n_t-1}{n_t}$ is required. This queuelength is the number of threads serviced at node m before a newly added thread in class i gets a service, so the waiting time for class i is computed accordingly. In Step 3, the cycle time for a thread in class i is the sum of waiting times at all nodes for a newly added thread in class i. Hence, the throughput for class i is $\frac{n_t}{cycle\ time\ for\ one\ thread}$. Step 4 computes new queuelengths at each node m, for class i threads when the population is N, using the throughput for class i and the waiting time for a class i thread at node m. Finally Step 5 verifies whether the difference between the queuelengths from successive iterations are within the tolerance level.

1- Initialize $\quad n^*_{i,m}(N) = \frac{N_i}{M}$

2- Calculate at each node m and for each class i

a- $\quad n^*_{i,m}(N-1_i) = 1 + \left[\frac{N_i-1}{N_i} n^*_{i,m}(N)\right] + \sum_{j=1,j\neq i}^{P} n^*_{j,m}(N)$

b- $\quad w^*_{i,m}(N) = \rho_{i,m}\left[n^*_{i,m}(N-1_i)\right]$

3- Evaluate for each class i

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

4- Evaluate the new values for $n^*_{i,m}(N)$ at all nodes m and for all classes i

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

5- If $difference(n^*_{i,m}(N)new,\ n^*_{i,m}(N)_{old}) > tolerance$
then go to *step 2* else *exit*

Figure 10: AMVA Algorithm

[7] This AMVA algorithm has been excerpted from Lazowska *et al* [15], and suitably modified for our work.

D Symbols and Their Meanings

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	
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.	2-16
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 $\Rightarrow$ dedicated node	
λ_{net}	Message rate from processor to the IN (message type)	
U_p	Processor (EU) utilization	
Analysis Related and Derived Parameters		
$em_{i,j}$	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 $\times$ 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	

Table 2: Model Parameters