PERFORMANCE ANALYSIS OF FINE–GRAIN MULTITHREADED MULTIPROCESSORS

W.M. ZUBEREK

Department of Computer Science
Memorial University
St.John's, Canada A1B 3X5
Email: wlodek@cs.mun.ca

Abstract. Instruction–level multithreading is an architectural approach to tolerating long–latency memory accesses and synchronization delays in distributed–memory systems. The paper presents a timed Petri net model of a fine–grain multithreaded distributed–memory multiprocessor system at the instruction execution level, and illustrates performance analysis by results obtained from simulation of the derived model. Performance equivalence is discussed in the context of multiprocessor systems with different interconnecting networks and different numbers of processors.

Keywords: fine–grain multithreading, distributed–memory architectures, timed Petri nets, performance analysis, discrete–event simulation.

1. INTRODUCTION

It is well known that if different components of a complex system are utilized at significantly different levels, the performance of the whole system is determined by a single component that has the highest utilization level. This is the component that first reaches its “saturation”, and limits the performance of the whole system. Such a component is called a system “bottleneck” [6].

Complex computer systems must address many such performance–related issues. Recent studies have shown that the number of processor cycles required to access main memory doubles approximately every six years [13]. This is due to the performance of microprocessors that has been doubling every 18 months (the so called Moore’s law [5]), while the performance of memory chips has been improving less than 10% per year [10]. In consequence, it is not unusual that as much as 60% of processor’s time is spent on waiting for the completion of memory operations [13].

In distributed–memory systems, the latency of memory accesses is even more pronounced than in centralized-memory systems because memory access requests may need to be forwarded through several intermediate nodes before they reach their destinations, and then the results need to be sent back to the original nodes. Each of the “hops” introduces some delay, which is typically assigned to the switches that control the traffic between the nodes [3, 4].

Instruction–level multithreading is a technique of tolerating long–latency memory accesses and synchronization delays in multiprocessor systems [1, 2], and in particular, in distributed–memory systems. The general idea of multithreading is quite straightforward. Since, in distributed–memory systems, a completion of a memory access can easily require hundreds or even thousands of processor cycles, the processor, when a long–latency memory operation occurs, instead of waiting for its completion, switches to another thread (if such a thread is ready for execution). If different threads are associated with different sets of processor registers, switching from one thread to another can be done very efficiently, requiring only a few processor cycles [1, 3].

In fine–grain multithreading, the thread is changed in every processor cycle [12, 16], so consecutive instructions are issued for execution from different threads. This approach is advantageous for eliminating data dependencies that stall the processor’s pipeline because consecutive instructions, issued from different threads, do not have data dependencies. Typically, the number of threads is equal to the number of (instruction) pipeline stages, so no two instructions from the same thread are executed at the same time. In fine–
grain multithreading, a thread issuing a long-latency memory operation becomes ‘waiting’ for the result of the requested operation. If a waiting thread is selected for instruction issuing, its slot simply remains empty (i.e., no instruction is issued), which is equivalent to a single-cycle pipeline stall. Since the threads issue their instructions one after another, fewer processor cycles are lost during a long-latency operation of a single thread.

A distributed memory system with 16 processors connected by a 2-dimensional torus-like network is used as a running example in this paper; an outline of such a system is shown in Fig.1.

It is assumed that all messages are routed along the shortest paths. It is also assumed that this routing is done in a nondeterministic way, i.e., if there are several shortest paths between two nodes, each of them is equally likely to be used. The average length of the shortest path between two nodes, or the average number of hops (from one node to another) that a message must perform to reach its destination, is usually determined assuming that the memory accesses are uniformly distributed over the nodes of the system.

Although many specific details refer to this 16-processor system, most of these details can easily be adjusted to other systems by changing the values of a few model parameters. For example, a 16-processor system can use a hypercube interconnecting network, as shown in Fig.2. It appears (as discussed in Section 4 in greater detail) that the two systems (Fig.1 and Fig.2) have very similar performance characteristics. However, two systems with 64 processors arranged in a 2-dimension torus-like network of 8 by 8 processors, and in a 6-th degree hypercube, have significantly different performance characteristics.

Each node in the network shown in Fig.1 (and Fig.2) is a fine-grain multithreaded processor which contains a processor, local memory, and two network interfaces, as shown in Fig.3. The outbound interface (or switch) handles outgoing traffic, i.e., requests to remote memories originating at this node as well as results of remote accesses to the memory at this node; the inbound interface handles incoming traffic, i.e., results of remote requests that “return” to this node and remote requests to access memory at this node.

The paper presents a timed Petri net model of a multithreaded distributed-memory system at the instruction execution level, and studies its performance. The model of a 16-processor system is introduced in Section 2 which also describes the main elements of the model. Section 3 presents performance results obtained by the simulation of the developed net model for different combinations of modeling parameters. Performance equivalence of different models is briefly discussed in Section 4. Section 5 concludes the paper.
2. TIMED PETRI NET MODEL

Petri nets have become a popular formalism for modeling systems that exhibit parallel and concurrent activities [11, 9]. In timed nets [15, 14], deterministic or stochastic (exponentially distributed) firing times are associated with transitions, and transition firings occur in real-time, i.e., tokens are removed from input places at the beginning of the firing period, and they are deposited to the output places at the end of this period.

A timed Petri net model of a fine-grain 4-threaded processor at the level of instruction execution is outlined in Fig.4, in which timed transitions are represented by solid bars and immediate transitions by thin bars.

The execution of each issued instruction is modeled by timed transition $Trun$ (in the center of Fig.4). Consecutive instructions are issued by consecutive threads; this is modeled by the thread control section between $Pent$ and $Pnxt$ (composed of 4 identical sections representing 4 threads). $Pend$ is a free-choice place with the choice probabilities reflecting the runlength, $\ell_t$, of threads (i.e., the average number of thread instructions executed between long-latency operations). In general, the free-choice probability assigned to $Pnxt$ is equal to $(\ell_t - 1)/\ell_t$, so if $\ell_t$ is equal to 10, the probability of choosing $Pnxt$ is 0.9; if $\ell_t$ is equal to 5, this probability is 0.8, and so on. The free-choice probability of $Tend$ is just $1/\ell_t$.

The (random) selection of $Tend$ for firing indicates a long-latency memory access issued by the current thread. The access request (to local or remote memory) is placed in $Mem$, and a token is also deposited in $Pwt$ to suspend the current thread (for the duration of the long-latency operation).

$Mem$ is a free-choice place, with a random choice of either accessing local memory ($Tloc$) or remote memory ($Trem$); in the first case, the request is directed to $Lmem$ where it waits for availability of Memory, and after accessing the memory, the thread returns to the queue of waiting threads, $Prd$. Memory is a shared place with two conflicting transitions, $Trmem$ (for remote accesses) and $Tlmem$ (for local accesses); the resolution of this conflict (if both requests are waiting) is based on marking-dependent (relative) frequencies determined by the numbers of tokens in $Lmem$ and $Rmem$, respectively.

The free-choice probability of $Trem$, $p_r$, is the probability of long-latency accesses to remote memory; the free-choice probability of $Tloc$ is $p_l = 1 - p_r$.

Requests for remote accesses are directed to $Rem$, and then, after a sequential delay (the outbound switch modeled by $Sout$ and $Tout$, forwarded to $Out$, where a random selection is made of one of the four (in this case) adjacent nodes; it is usually assumed that the information is uniformly distributed over the node of the system, so all nodes are selected with equal probabilities. Similarly, the incoming traffic is collected from all neighboring nodes in $Inp$, and, after a sequential delay (the inbound switch $Sinp$ and $Tsinp$), forwarded to $Dec$. $Dec$ is a free-choice place with three transitions sharing it: $Tret$, which represents the satisfied requests returning to their ‘home’ nodes; $Tgo$, which represents requests as well as responses forwarded to another node (another ‘hop’ in the interconnecting network); and $Tlocal$, which represents remote requests accessing the memory at the destination node; these remote requests are queued in $Rmem$ and served by $Trmem$ when the $Memory$ becomes available. The free-choice probabilities associated with $Tret$, $Tgo$ and $Tlocal$ characterize the interconnecting network [4].

The traffic outgoing from a node (place $Out$) is composed of requests and responses forwarded to another node (transition $Tg$, requests to requests from other nodes (transition $Trmem$) and remote memory requests originating in this node (transition $Trem$).

3. PERFORMANCE ANALYSIS

The parameters which characterize the model of the fine-grain multithreaded distributed-memory system include:

- $n_p$ – the number of processors,
- $n_t$ – the number of threads,
- $\ell_t$ – the thread runlength,
- $t_p$ – the processor cycle time,
- $t_r$ – the memory cycle time,
- $t_s$ – the switch delay,
- $n_h$ – the average number of hops,
- $p_l$ – the probability to access local memory,
- $p_r$ – the probability to access remote memory,
- $p_t = 1 - p_r$.

For performance analysis, it is convenient to represent all timing information in relative rather than absolute units, and the processor cycle, $t_p$, has been as-
sumed as the unit of time. Consequently, all temporal data are expressed in processor cycles; e.g., $t_m = 10$ means that the memory cycle time ($t_m$) is equal to 10 processor cycles, $t_s = 5$ means that the switch delay ($t_s$) is equal to 5 processor cycles.

An example of the utilization of processors in a 16–processor system, as a function of $p_L$, the probability of long–latency accesses to local memory, is shown in Fig.5, where plot (1) corresponds to a 4–thread processor (as in Fig.4), and plot (2) to an 8–thread processor. It can be observed that the number of threads has a rather insignificant influence on the performance. On the other hand, the utilization deteriorates for smaller values of $p_L$, i.e., when an increasing number of memory accesses use the interconnecting network. This may indicate that the switches in the interconnecting network are the system bottleneck.

The utilization of the input switch, in a 4–thread and
an 8–thread system, also as a function of \( p \ell \), is shown in Fig.6. The region of low utilization of processors in Fig.5 (i.e., the region of small values of \( p \ell \)), corresponds to almost 100% utilization of the input switches, which indicates that the switches are the system bottlenecks; the switches are simply too slow for this system.

In general, system bottlenecks can be identified by analyzing service demands for the components of the system; the component with the highest service demand is the bottleneck because it will first reach the upper limit of its utilization. For the component \( i \), the service demand \( d_{i} \) is the product of the rate of requests (sometimes also called the ‘visit rate’), \( v_{i} \), and the (average) service time of this component, \( s_{i} \), i.e.,

\[
d_{i} = v_{i} \times s_{i}.
\]

In the model of the multithreaded multiprocessor system described in the previous section, the components are (\( n_p \) is the number of processors):

- processors with service demands \( d_{p,j} \), \( j = 1, \ldots, n_p \);
- memories with service demands \( d_{m,j} \), \( j = 1, \ldots, n_p \);
- inbound network switches with service demands \( d_{si,j} \), \( j = 1, \ldots, n_p \);
- outbound network switches with service demands \( d_{so,j} \), \( j = 1, \ldots, n_p \).

If all processors are identical, the steady–state service demands at all nodes are identical, so the second subscripts can be dropped.

In the steady–state, service demands can be considered for one (average) cycle of thread execution and suspension, taking into account the demands of all \( n_t \) threads. The service demand for the processor is the product of thread runlength, \( \ell_t \), the number of threads, \( n_t \), and the processor cycle time, \( t_p \).

The service demand for the memory subsystem has two components, one due to local memory requests and the other due to requests coming from remote processors. The component due to local requests is the product of the visit rate (which is the probability of local accesses), \( p \ell \), the number of threads, \( n_t \), and the memory cycle time, \( t_m \). Likewise, the component due to remote accesses is \( p_r \times n_h \times t_m \); this expression is obtained by taking into account that, for each node, the requests are coming from \( (n_p - 1) \) other processors, and that remote memory requests are uniformly distributed over \( (n_p - 1) \) processors, so the service demand due to remote requests is \( p_r \times n_h \times t_m \times (n_p - 1)/(n_p - 1) = p_r \times n_h \times t_m \).

The service demand for the inbound switch (in each processor) can be obtained as follows. The visit rate to an inbound switch (due to a single processor) is the product of probability of remote accesses, \( p_r \), the number of threads, \( n_t \), the average number of hops (in both directions), \( 2 \times n_h \), and the switch delay, \( t_s \). Remote memory requests from all \( n_p \) processors are distributed across the \( n_p \) inbound switches, so the service demand for an inbound switch from a single thread is \( 2 \times p_r \times n_h \times t_s \times n_p/n_p = 2 \times p_r \times n_t \times n_h \times t_s \). For the outbound switch, the service demand is \( 2 \times t_s \times p_r \times n_t \); the number of hops, \( n_h \), does not affect this service demand.

The service demands are thus:
The value of $n_h$ can be determined by analyzing the numbers of hops required to access information uniformly distributed over the nodes of the system. For the 16-processor system outlined in Fig.1, the numbers of hops (from the “reference node”) are shown in Fig.7 (the “reference node”, which can be any node in the system, is indicated as the “dark” node).

There are 4 nodes in the distance of 1 hop from the reference node, 6 nodes in the distance of 2 hops, 4 nodes in the distance of 3 hops and just 1 node in the distance of 4 hops; the average number of hops is thus:

$$n_h^{(16)} = \frac{4 \cdot 1 + 6 \cdot 2 + 4 \cdot 3 + 1 \cdot 4}{15} \approx 2.$$

For other torus-like interconnection networks, the value of $n_h$ can be approximated reasonably well by $\sqrt{n_p}/2$ [4], where $n_p$ is the number of processors.

In a complex system, the utilizations of components is directly related to service demands for these components. If the service demands for all components of a system are equal, the system is called balanced [6] as the utilizations of all components are also equal. For the 16-processor system (as in Fig.1), if $\ell_t = t_m$ ($t_m$ expressed in processor cycles), the system is balanced when $4 \cdot p_r \cdot t_s = \ell_t$. If also $t_s = \ell_t$ (as in Fig.5 and Fig.6), the balancing condition is $p_r = 0.25$ (or $p_T = 0.75$). This value corresponds to the “knee” of the switch utilization plots (Fig.6), and also corresponds to the beginning of the region of reduced utilization of processors (Fig.5).

Fig.8 and Fig.9 show the utilization of processors and switches, as in Fig.5 and Fig.6, but for the case when the switch delay is one half of that used in Fig.5 and Fig.6. Consequently, the processor utilization is generally much better, and is reduced only in the region of small values of $p_T$ in which the utilization of switches (Fig.9) remains high (in this region the switches remain the bottlenecks). Further reduction of the switch delay would result in further improvement of the utilization of processors, but this further improvement cannot be significant.
essence of the concept of performance equivalence [18].

Performance equivalent systems can be used to simplify performance analysis of multiprocessor systems (as well as other systems which have a similar structure). More specifically, since the simulation time required for simulation-based performance analysis of multiprocessor systems depends (superlinearly) upon the number of processors, instead of simulating a system containing \( n_p \) processors, a much simpler performance equivalent system can be used, significantly reducing the required simulation time, and providing reasonably accurate results. For performance analysis of the 16–processor system (Fig.1), a 4–processor system can be used with the same parameters \( \ell_t \) and \( t_m \), and with the switch delay \( t_s^{(4)} \) adjusted to the value which compensates the differences in the values of \( n_h \) between the 16–processor, \( n_h^{(16)} \), and 4–processor, \( n_h^{(4)} \), systems, i.e., such that:

\[
n_h^{(16)} \times t_s^{(16)} = n_h^{(4)} \times t_s^{(4)}.
\]

Since \( n_h^{(4)} = 4/3 \), performance equivalence is obtained for \( t_s^{(4)} = 1.5 \times t_s^{(16)} \).

In a 16–processor hypercube network (Fig.2), each node is connected to 4 other nodes as in the 2–dimension torus–like network (Fig.1). The values of \( n_h \) for these two networks are thus the same, and the two 16–processor systems, built of the same components, are performance equivalent.

For a 64–processor system with a hypercube interconnecting network, the average number of hops is equal to 3; such a system is performance equivalent to a 16–processor system with a two–dimensional torus–like interconnecting network in which the switch delays are 1.5 greater than those in the 64–processor system, and also to a 4–processor system in which the switch delays are 2.25 greater than those in the 64–processor system. A 64–processor system with the hypercube interconnection network is performance equivalent to a 64–processor system with a two–dimensional torus–like network if the switch delays of the latter network are 0.75 of the switch delays of the hypercube system; on the other hand, the 64–processor hypercube system is performance equivalent to a 64–processor system with a three–dimensional torus–like interconnecting network with the same switch delays (since the value of \( n_h \) for such a network is also equal to 3), and so on.

5. CONCLUDING REMARKS

The paper presents a timed Petri net model of fine-grain multithreaded multiprocessor system at the instruction execution level, and analyzes its performance, and in particular, the effects of system bottlenecks on the performance of the system. System bottlenecks are identified by analyzing service demands for the components of the system. Removing the bottlenecks (or balancing the system) can significantly improve its performance; reducing the delay of switches in the original system (the bottleneck) practically doubles the utilization of processors in the critical region of small values of \( p_t \) (Fig.5 and Fig.8).

Balancing the system by improving performance characteristics of its components may sometimes be difficult because the components with improved characteristics may not be available. There is, however, an alternative solution; an improved performance can be obtained by replicating the components and using several identical components working concurrently and sharing the workload; simulation studies [17] indicate that such a solution is practically as efficient as the components with improved performance characteristics.

Since the utilization of processors is probably the simplest indicator of the performance of the whole system, there may be a tendency to maximize this utilization. The simplest way to achieve this is to make the processor the bottleneck. Fig.10 shows the utilization of processors for the same 16–processor system (as in Fig.1), but for the case when both \( t_m \) and \( t_s \) are equal to one half of \( \ell_t \).

\[
\ell_t = 10, t_m = 5, t_s = 5.
\]

Fig.10. Processor utilization, (1) 4 threads, (2) 8 threads;
The processor’s utilization is reasonably high (60% to 80%), and is almost independent of the probability to access local memory (the switch becomes the bottleneck only for \( p_L \) very close to 0).

Although the discussion and presented results refer to a 16–processor system, the model needs only a few small changes to represent other multiprocessor systems. For example, the only changes that need to be made to represent a 25–processor or a 36–processor system, are the values of the free–choice probabilities associated with the transitions of \( \text{Dec} \) (to represent different values of \( n_H \)).

The models of fine–grain multithreaded multiprocessors contain many identical submodels representing individual threads and individual processors. The model can be significantly reduced using high–level net models, and in particular, colored Petri nets [7, 8], in which all repetitions of the same submodel can be eliminated and replaced by token attributes called “colors”. A complete model of a multithreaded multiprocessor system (with any number of processors and any number of threads, and even with different topologies of interconnecting networks) is shown in Fig.11 (all color–related descriptions are not shown in Fig.11; in particular, it should be understood that each token shown in Fig.11 actually represents \( n_p \) tokens with different colors denoting individual processors of the system). The net model is quite simple, but its analysis becomes much more complex than in the case of “standard” net models (as in Fig.4).

Acknowledgement

The Natural Sciences and Engineering Research Council of Canada partially supported this research through grant RGPIN-8222.

REFERENCES


BIOGRAPHY

W.M. ZUBEREK received M.Sc. degree in Electronic Engineering and Ph.D. and D.Sc. degrees in Computer Science, all from Warsaw University of Technology. Currently he is a Professor in the Department of Computer Science of Memorial University of St.John’s, Canada, and the Chair of a new Interdisciplinary Computational Science Program. His research interests include modeling and performance analysis of concurrent systems, and in particular applications of timed Petri nets, hierarchical modeling and discrete–event simulation to analysis of complex systems. His home Web page is www.cs.mun.ca/~wlodek.