PERFORMANCE EQUIVALENCE IN THE SIMULATION OF MULTIPROCESSOR SYSTEMS

W.M. ZUBEREK

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

Abstract. In simulation–based performance evaluation, the simulation time is directly related to the complexity of the simulated systems. Since modern multiprocessor systems contain hundreds and even thousands of processors, simulation of such systems can be quite time–demanding. This paper studies multiprocessor systems with different numbers of processors but with the same utilizations of corresponding components; such systems are called performance equivalent. Performance equivalence can be used to simplify simulation–based performance analysis of complex systems by simulating much simpler systems which are equivalent with respect to performance to the original ones. It is shown that in some cases identifying performance equivalent systems is quite straightforward.

Keywords: multiprocessor systems, timed Petri nets, performance equivalence, discrete–event simulation.

1. INTRODUCTION

Multiprocessor systems are usually classified as shared–memory systems or distributed–memory systems [8]. Shared–memory systems can have uniform access to the (shared) memory (typically centralized, as in bus–based systems or systems with multistage connecting networks or crossbar switches), or the latencies of memory accesses may differ, as is typical for distributed (shared on non–shared) memory systems. In distributed–memory systems, the (total) memory is composed of modules physically located at different nodes of the system, so the latency depends upon the node which originates the request for memory access as well as the node which contains the requested information. Shared–memory systems are believed to be easier to program, but distributed–memory systems scale in a better way; systems with large numbers of processors are usually distributed–memory systems [17].

The performance of a distributed–memory system depends upon a number of factors. The probability of requesting an access to a remote node is one of important parameters; if this probability is close to zero, the nodes can be analyzed in isolation from each other, which significantly simplifies the evaluation. If this probability cannot be neglected, the analysis needs to be performed for the complete multiprocessor system since the behavior of the interconnecting network, and in particular, its congestion, affects the performance of each node of the system. If the system is composed of identical nodes, the steady–state behavior of all nodes can be assumed identical, which creates symmetries that can, sometimes, be used for simplification of the evaluation process. For simulation–based performance evaluation, however, such symmetries cannot easily be taken into account, so a model of the complete system needs to be simulated.

The purpose of this paper is to study simplifications of the simulation–based performance evaluation of distributed–memory multiprocessor systems. The simplifications are based on performance equivalence of systems. More specifically, the paper studies multiprocessor systems which are composed of identical processors, but which have different numbers of processors, yet all corresponding components (such as processors, memories, or interconnects) have the same utilizations. Such systems are called equivalent with respect to performance or performance equivalent. Performance equivalence can be used to reduce the simulation time required for simulation–based performance analysis because instead of the original complex system, a much simpler system can be analyzed providing a good approximation of the results for the original, complex system. The paper also shows that for some systems such performance

equivalence is quite straightforward to establish.

A multiprocessor system with 16 processors connected by a 2–dimensional torus–like network, shown in Fig.1 [19], is used as a running example in this paper.

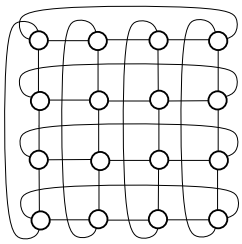

Fig.1. Outline of a 16–processor system.

In distributed–memory systems, it is usually assumed that the memory access requests sent from one node to another 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 shortests paths between two nodes, each of them is equally likely to be used. Consequently, the traffic is assumed to be uniformly distributed in the interconnecting network. The average length of the shortest path between two nodes, or the average number of hops (from one node to another) that a request must perform to reach its destination, n_h, is usually determined assuming that the memory accesses are uniformly distributed over the nodes of the system.

In modern computer systems, the performance of memory is increasingly often becoming the factor limiting the performance of the system. Due to continuous progress in manufacturing technologies, the performance of processors has been doubling every 18 months (the so–called Moore's law [7]). However, the performance of memory chips has been improving by only 10% per year [14], creating a "performance gap" in matching processor's performance with the required memory bandwidth. In effect, it is becoming more and more often the case that the performance of applications depends on the performance of machine's memory hierarchy [3].

Memory hierarchies, and in particular multi–level cache memories, have been introduced to reduce the effective latency of memory accesses. Cache memories provide efficient access to information when the information is available at lower levels of memory hierarchy; occasionally, however, long–latency memory operations are needed to transfer the information from the higher levels of memory hierarchy to the lower ones. Extensive research has focused on reducing and tolerating these large memory access latencies. Techniques for reducing the frequency and impact of cache misses include hardware and software prefetching [5, 10], speculative loads and execution [15] and, increasingly often, multithreading [1, 4].

Instruction–level multithreading, and in particular block–multithreading [1, 2, 4], tolerates long–latency memory accesses and synchronization delays by switching the threads rather than waiting for the completion of a long–latency operation which, in a distributed–memory system, can require thousands of processor cycles. A combination of multithreading and superscalar architecture is also an approach used in high–performance microprocessors [11].

Each node in the system shown in Fig.1 is assumed to be a multithreaded processor, local memory, and two network interfaces, as shown in Fig.2.

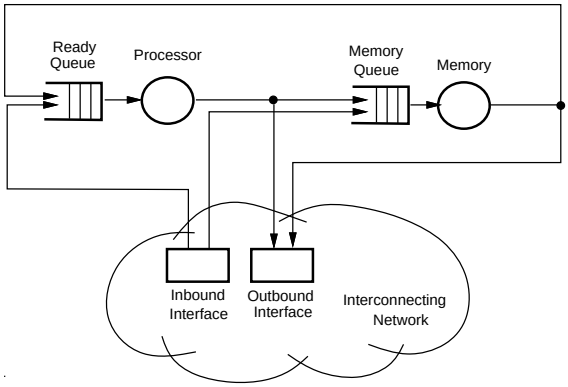

Fig.2. Outline of a multithreaded processor.

The outbound 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.

Fig.2 also shows a queue of ready threads; whenever the processor performs a context switching (i.e., switches from one thread to another), a thread from this queue is selected for execution and the execution continues until another context switching is performed. In block multithreading, context switching is performed for all long–latency memory accesses by 'suspending' the current thread, forwarding the memory access request to the relevant memory module (lo-

cal, or remote using the interconnecting network) and selecting another thread for execution. When the result of this request is received, the status of the thread changes from 'suspended' to 'ready', and the thread joins the queue of ready threads, waiting for another execution phase on the processor.

The average number of instructions executed between context switching is called the runlength of a thread, ℓ_t, which is one of main modeling parameters. It is directly related to the probability that an instruction requests a long–latency memory operation.

Another important modeling parameter is the probability of long–latency accesses to local, p_ℓ, (or remote, $p_r = 1 - p_\ell$) memory (in Fig.2 it corresponds to the "decision point" between the *Processor* and the *Memory Queue*); as the value of p_ℓ decreases (or p_r increases), the effects of communication overhead and congestion in the interconnecting network (and its switches) become more pronounced; for p_ℓ close to 1, the nodes can be practically considered in isolation.

The (average) number of available threads, n_t, is yet another basic modeling parameter. For very small values of n_t, queueing effects can be practically neglected, so the performance can be predicted by taking into account only the delays of system's components. On the other hand, for large values of n_t, the system can be considered in saturation, which means that one of its components will be utilized in almost 100 %, limiting the utilization of other components as well as the whole system. Identification of such limiting components (called the bottlenecks [9]) and improving their performance is the key to the improved performance of the entire system.

Section 2 introduces a timed Petri net model of a multithreaded multiprocessor system. Performance equivalence is defined in Section 3, while Section 4 presents results illustrating the practical use of performance equivalence. Several concluding remarks are given in Section 5.

2. MODEL OF A MULTIPROCESSOR SYSTEM

Petri nets have become a popular formalism for modeling systems that exhibit parallel and concurrent activities [13, 12]. In order to take the durations of these activities into account, several types of Petri nets *with time* have been proposed by assigning *firing times* to the transitions or places of a net. In timed nets [18, 16], 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 multithreaded processor at the level of instruction execution is shown in Fig.3 [19]. As usual, timed transitions are represented by "thick" bars, and immediate ones, by "thin" bars.

The execution of each instruction of the 'running' thread is modeled by transition $Trun$, a timed transition with the firing time representing one processor cycle, t_p. Place $Proc$ represents the (available) processor (if marked) and place *Ready* – the queue of threads waiting for execution. The initial marking of *Ready* represents the (average) number of available threads, n_t.

If the processor is available (i.e., $Proc$ is marked) and $Ready$ is not empty, a thread is selected for execution by firing the immediate transition $Tsel$. Execution of consecutive instructions of the selected thread is performed in the loop $Pnxt$, $Trun$, $Pend$ and $Tnxt$. $Pend$ is a free–choice place with the choice probabilities determined by the runlength, ℓ_t, of the thread. In general, the free–choice probability assigned to $Tnxt$ is equal to $(\ell_t - 1)/\ell_t$, so if ℓ_t is equal to 10, the probability of $Tnxt$ is 0.9; if ℓ_t is equal to 5, this probability is 0.8, and so on. The free–choice probability of $Tend$ is just $1/\ell_t$.

If $Tend$ is chosen for firing rather than $Tnxt$, the execution of the thread ends, a request for a long–latency access to (local or remote) memory is placed in Mem, and a token is also deposited in *Pcsw*. The timed transition $Tcsw$ represents the context switching and is associated with the time required for the switching to a new thread, t_{cs}. When its firing is finished, another thread is selected for execution (if it is available).

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 ($Tlmem$), the thread returns to the queue of waiting threads, $Ready$. *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.

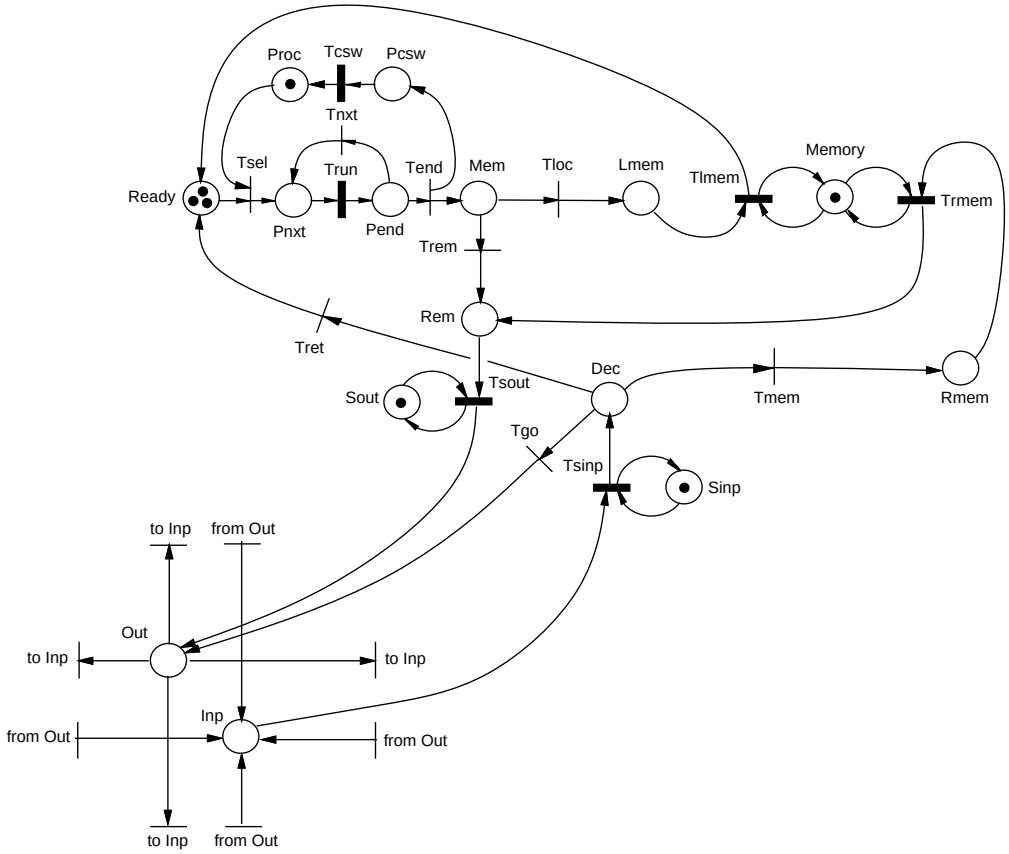

Fig.3. Petri net model of a multithreaded processor at the instruction execution level.

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 $p_\ell = 1 - p_r$.

Requests for remote accesses are directed to Rem, and then, after a sequential delay (the outbound switch modeled by $Sout$ and $Tsout$), forwarded to Out, where a random selection is made of one of the four (in this case) adjacent nodes (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 reaching their "home" nodes; Tgo, which represents requests as well as responses forwarded to another node (another 'hop' in the interconnecting network); and $Tmem$, 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 module *Memory* becomes available. The free–choice probabilities associated with $Tret$, Tgo and $Tmem$ characterize the interconnecting network [6]. For a 16–processor system (as in Fig.1), and for memory accesses uniformly distributed among the nodes of the system, the free–choice probabilities of $Tmem$ and Tgo are 0.5 for forward moving requests, and 0.5 for $Tret$ and Tgo for returning requests.

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

3. PERFORMANCE EQUIVALENCE

The utilizations of components in complex systems is directly related to service demands for these components; a system is called balanced if the service demands for all components are equal [9]. The service demand, d_i, of the component 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., $d_i = v_i * s_i$.

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

- processors with service demands $d_{p,j}$, $j = 1, ..., N$;
- memories with service demands $d_{m,j}$, $j = 1, ..., N$;
- inbound network switches with service demands $d_{si,j}$, $j = 1, ..., N$;
- outbound network switches with service demands $d_{so,j}$, $j = 1, ..., N$.

Because of the symmetries of the system, in the steady–state the service demands at all nodes are identical, so the second subscript can be dropped. The description of the service demands uses the following parameters of the model:

parameter	*symbol*
thread runlength	ℓ_t
processor cycle time	t_p
memory cycle time	t_m
switch delay	t_s
average number of hops	n_h
prob. to access local memory	p_ℓ
prob. to access remote memory	$p_r = 1 - p_\ell$

For a single cycle of state changes of a thread (i.e., a thread going through the phases of execution, suspension, and then waiting for another execution), the service demand for the processor is the product of thread runlength, ℓ_t, and 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_ℓ, and memory cycle, t_m. Likewise, the component due to remote accesses is $p_r * t_m$; this expression is obtained by taking into account that for each node the requests are coming from $(N-1)$ remote processors, and that remote memory requests are uniformly distributed over $(N-1)$ processors, so the service demand due to remote requests is $p_r * t_m * (N-1)/(N-1) = p_r * t_m$.

The service demand for the inbound switch due to a single thread (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, average number of hops (in both directions), $2 * n_h$, and the switch delay, t_s. Remote memory requests from all N processors are distributed across the N inbound switches, so the service demand for an inbound switch due to a single thread is $2 * p_r * n_h * t_s * N/N = 2 * p_r * n_h * t_s$. For the outbound switch, the service demand is $2 * t_s * p_r$; the number of hops, n_h, does not affect this service demand.

The service demands are thus:

$$\begin{aligned} d_p &= \ell_t * t_p; \\ d_m &= p_\ell * t_m + p_r * t_m = t_m; \\ d_{si} &= 2 * p_r * n_h * t_s; \\ d_{so} &= 2 * p_r * t_s. \end{aligned}$$

Let D denote the maximum component service demand, $D = \max(d_p, d_m, d_{si}, d_{so})$. Relative service demands are the service demands divided by D, d_p/D, d_m/D, etc.

Two systems are equivalent with respect to performance (or performance equivalent) if the relative service demands for all their corresponding components are the same. A straightforward consequence of this definition is that component utilizations in performance equivalent systems are the same; this is the essense of the concept of performance equivalence. A straightforward example of two systems which are performance equivalent is a pair of systems with the same rate of all corresponding (average) service times; a system and its replacement in which the service times of all components are reduced at the same rate could be an illustration of this concept. However, many other systems can also be performance equivalent.

Performance equivalent systems can be used to simplify performance analysis of distributed–memory multithreaded 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 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 ℓ_t and t_m, and with the switch delay t_s adjusted to the value which compensates the differences in the values of n_h between the 16–processor and 4–processor systems, i.e., such that:

$$n_h^{(16)} * t_s^{(16)} = n_h^{(4)} * t_s^{(4)}.$$

Since $n_h^{(16)}$ can be approximated reasonably well by 2 [6], and $n_h^{(4)} = 4/3$, performance equivalence is obtained for $t_s^{(4)} = 1.5 * t_s^{(16)}$.

4. PERFORMANCE RESULTS

Performance results discussed in this section assume that all timing characteristics are expressed in processor cycles (which is assumed to be 1 unit of time).

Fig.4 shows the utilization of the processors, in a 16–processor system, as a function of the number of available threads, n_t, and the probability of long–latency accesses to local memory, p_ℓ, for fixed values of other parameters.

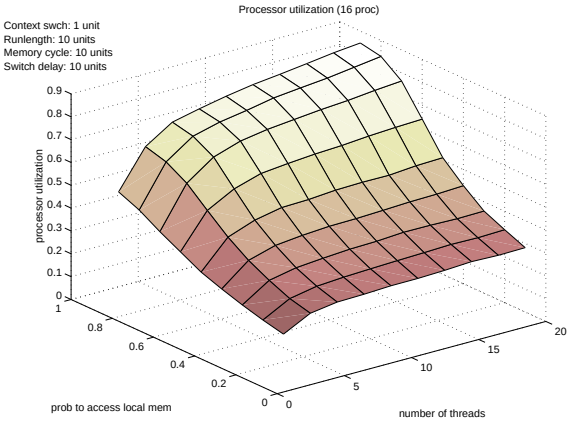

Fig.4. Processor utilization – 16 processors; $t_{cs} = 1, \ell_t = 10, t_m = 10, t_s = 10$.

The utilization of the processors in a performance equivalent 4–processor system is shown in Fig.5; performance equivalence in the 4–processor system is obtained by using the same values of parameters ℓ_t and t_m, and increasing t_s to 15 processor cycles to compensate for the decreased value of n_h.

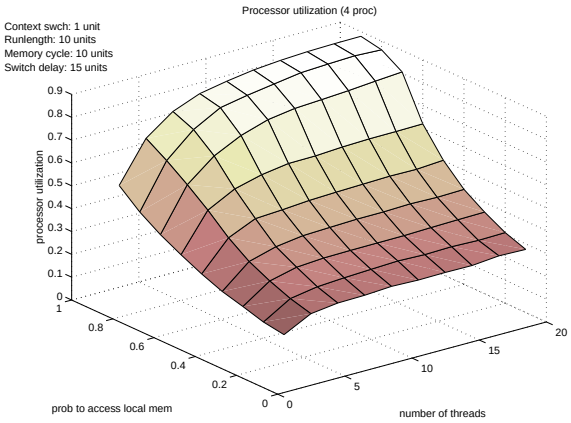

Fig.5. Processor utilization – 4 processors; $t_{cs} = 1, \ell_t = 10, t_m = 10, t_s = 15$.

It can be observed that the results are fairly similar, with differences not exceeding a few percent.

For smaller values of p_ℓ (or larger value of p_r), i.e., when an increasing number of memory accesses is to remote memory, the utilization of processors decreases significantly in Fig.4 and Fig.5. This is an indication that the interconnection network, and more specifically, the switches are the bottleneck in this system (i.e., they are utilized in almost 100%, limiting the performance of the entire system). Indeed, Fig.6 shows the utilization of the (input) switches in the 16–processor system as a function of the number of available threads, n_t, and the probability of long–latency accesses to remote (not local) memory, p_r (so the front part of Fig.6 corresponds to the back part of Fig.4). Fig.7 shows the same switch utilization in a performance equivalent 4–processor system.

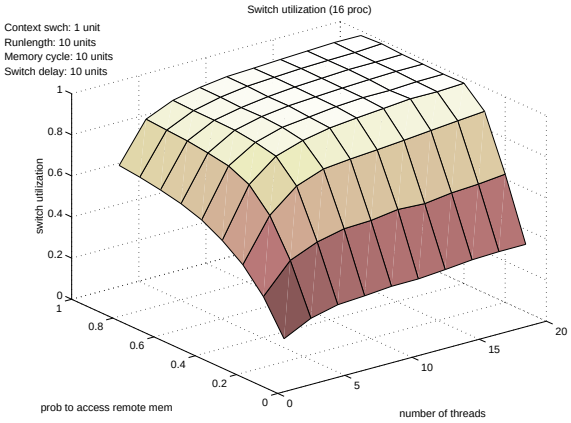

Fig.6. Switch utilization – 16 processors;

$t_{cs} = 1, \ell_t = 10, t_m = 10, t_s = 10.$

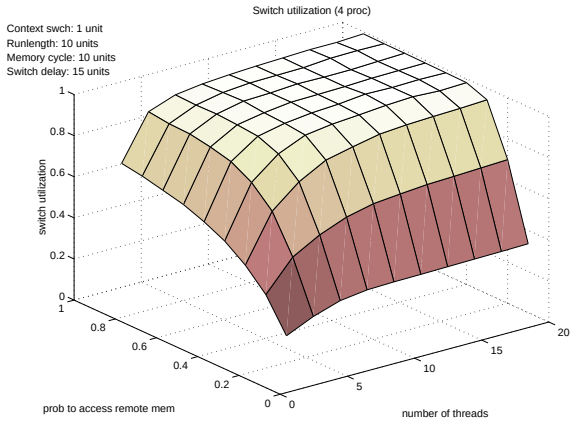

Fig.7. Switch utilization – 4 processors;
$t_{cs} = 1, \ell_t = 10, t_m = 10, t_s = 15.$

Fig.6 and Fig.7 show that with the exception of small values of p_r and small values of n_t, the utilization of the switches in both systems, the 16–processor one and its performance equivalent 4–processor system, is almost 100%. This is a clear indication that the switches are simply to slow for these systems; any reduction of the switch delay will result in improved utilization of the processors.

Fig.8 and Fig.9 show the utilization of the processors in the 16–processor system and in a performance equivalent 4–processor system, for the case when the switch delay is reduced to one half of its original value.

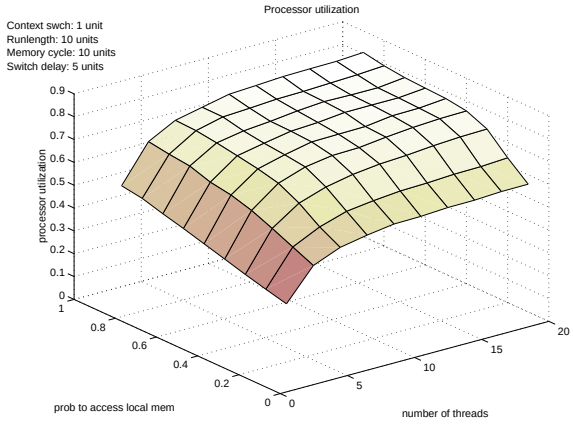

Fig.8. Processor utilization – 16 processors;
$t_{cs} = 1, \ell_t = 10, t_m = 10, t_s = 5.$

As before, the agreement of the results obtained for the original 16–processor model and its performance

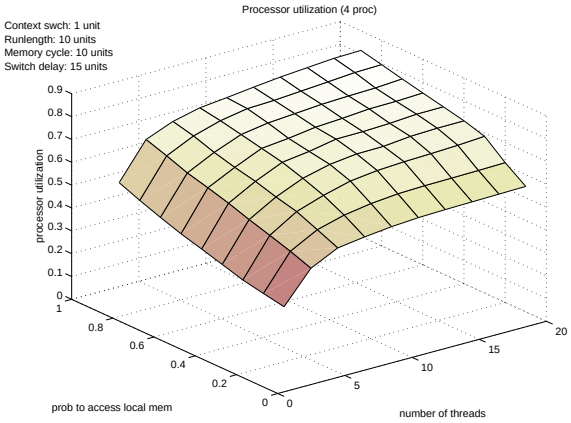

Fig.9. Processor utilization – 4 processors;
$t_{cs} = 1, \ell_t = 10, t_m = 10, t_s = 7.5.$

equivalent 4-processor system, is quite good. Moreover, processor utilizations in Fig.8 and Fig.9 are significantly better than in Fig.4 and Fig.5; the region in which the switch is the bottleneck is substantially reduced, as shown in Fig.10 for the 16–processor system. Any further reduction of the switch delay (or increase in the switch throughput that can be obtained by using several parallel switches [21]), will further improve the utilization of processors, but these gains will be restricted to regions of small values of p_r and n_t; the overhead of context switching introduces an upper bound on the utilization of processors at the level of $\ell_t/(\ell_t + t_{cs})$.

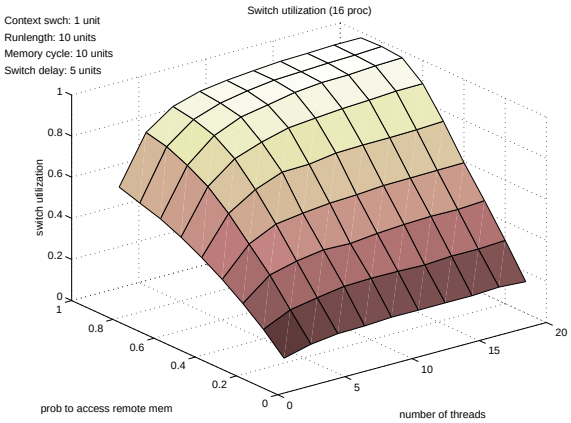

Fig.10. Switch utilization – 16 processors;
$t_{cs} = 1, \ell_t = 10, t_m = 10, t_s = 5.$

5. CONCLUDING REMARKS

The presented performance results for distributed–memory multithreaded multiprocessor systems indicate that significant simulation–time reductions can

be achieved by using simpler models which are equivalent with respect to performance to the original systems. Since the simulation time of complex models usually increases superlinearly with the size of the model, the gains in the simulation time also increase more than linearly with the size of the (original) model.

A slightly different approach to performance equivalence is presented in [20] where instead of changind the delays of switches, the net model is modified in such a way that the value of n_h is preserved at the level of the original system, independently of the number of processors, so, again, much simpler model can be simulated to reduce the simulation time.

The results obtained for a 2–dimensional torus–like network are also valid for other interconnecting networks with the same connectivity characteristics. For example, Fig.11 shows a hypercube network for a 16–processor system that is composed of two 8–processor subsystems. Since the average number of hops in this network is the same as in the two–dimensional network shown in Fig.1, the performance characteristics of both networks are also the same (although the two interconning networks scale in different ways).

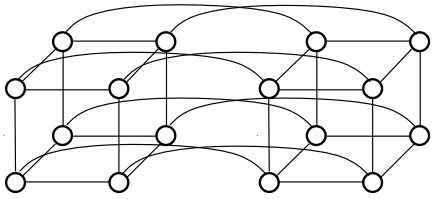

Fig.11. Outline of a 16–processor system.

The derived models assume that accesses to memory are uniformly distributed over the nodes of the system. If this assumption is not realistic and some sort of 'locality' is present, the only change that needs to be done is an adjustment of the value of n_h; for example, if the probability of accessing nodes decreases with the distance (i.e., nodes which are close are more likely to be accessed that the distant ones), the value of n_h will be smaller than that determined for the uniform distribution of accesses, and will result in improved performance.

It should be noted that the presented results provide only some insight into the behavior of multithreaded systems as some of the assumptions are not satisfied in the real systems – for example, the number of threads is rarely constant, the probabilities of accessing local or remote memory may change during the executions of programs, and so on.

ACKNOWLEDGMENT

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

REFERENCES

[1] Agarwal, A., "Performance tradeoffs in multithreaded processors"; *IEEE Trans. on Parallel and Distributed Systems*, vol.3, no.5, pp.525-539, 1992.

[2] Boothe, B. and Ranade, A., "Improved multithreading techniques for hiding communication latency in multiprocessors"; *Proc. 19-th Annual Int. Symp. on Computer Architecture*, Gold Coast, Australia, pp.214-223, 1992.

[3] Burger, D., Goodman, J.R., Kaegi, A., "Memory bandwidth limitations of future microprocessors"; *Proc. 23-rd Annual Int. Symp. on Computer Architecture*, Philadelphia, PA, pp.78-89, 1996.

[4] Byrd, G.T. and Holliday, M.A., "Multithreaded processor architecture"; *IEEE Spectrum*, vol.32, no.8, pp.38-46, 1995.

[5] Chen, T-F. and Baer, J-L., "A performance study of software and hardware data prefetching scheme"; *Proc. 21-st Annual Int. Symp. on Computer Architecture*, Chicago, IL, pp.223-232, 1994.

[6] Govindarajan, R., Suciu, F. and Zuberek, W.M., "Timed Petri net models of multithreaded multiprocessor architectures"; *Proc. 7-th Int. Workshop on Petri Nets and Performance Models*, St. Malo, France, pp.153-162, 1997.

[7] Hamilton, S., "Taking Moore's law into the next century"; *IEEE Computer Magazine*, vol.32, no.1, pp.43-48, 1999.

[8] Hwang, K., *Advanced computer architecture – parallelism, scalability, programmability*; McGraw-Hill 1993.

[9] Jain, R., *The art of computer systems performance analysis*; J. Wiley & Sons 1991.

[10] Klaiber, A.C. and Levy, H.M., "An architecture for software-controlled data prefetching"; *Proc. 18-th Annual Int. Symp. on Computer Architecture*, Toronto, Canada, pp.43-53, 1991.

[11] Loh, K.S. and Wong, W.F., "Multiple context multithreaded superscalar processor architecture"; *Journal of Systems Architecture*, vol.46, pp.243-258, 2000.

[12] Murata, T., "Petri nets: properties, analysis and applications"; *Proceedings of IEEE*, vol.77, no.4, pp.541–580, 1989.

[13] Reisig, W., "Petri nets - an introduction" (EATCS Monographs on Theoretical Computer Science 4); Springer–Verlag 1985.

[14] Rixner, S., Dally, W.J., Kapasi, U.J., Mattson, P. and Ovens, J.D., "Memory access scheduling"; *Proc. 27-th Annual Int. Symp. on Computer Architecture*, Vancouver, Canada, pp.128-138, 2000.

[15] Rogers, A. and Li, K., "Software support for speculative loads"; *Proc. 5-th Symp. on Architectural Support for Programming Languages and Operating Systems*, pp.38-50, 1992.

[16] Wang, J., *Timed Petri nets – theory and application*; Kluwer Academic Publ. 1998.

[17] Wilkinson, B., *Computer architecture – design and performance*; Prentice Hall 1996.

[18] Zuberek, W.M., "Timed Petri nets – definitions, properties and applications"; *Microelectronics and Reliability* (Special Issue on Petri Nets and Related Graph Models), vol.31, no.4, pp.627–644, 1991.

[19] Zuberek, W.M., "Performance modeling of multithreaded distributed memory architectures", *Proc. 2-nd Workshop on Hardware Design and Petri Nets*, Williamsburg, VA, pp.63–82, 1999.

[20] Zuberek, W.M., "Approximate simulation of distributed-memory multithreaded multiprocessors"; *Proc. 35-th Annual Simulation Symposium*, San Diego, CA, pp.107–114, 2002.

[21] Zuberek, W.M., "Analysis of performance bottlenecks in multithreaded multiprocessor systems"; *Fundamenta Informaticae*, vol.50, no.2, pp.223–241, 2002.

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 in 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`.