2009 Winter Simulation Conference – Modeling Methodology Track, Austin, Texas, December 13-16, 2009, pp.899-907.
Copyright © 2009 IEEE (DOI 10.1109/WSC.2009.5429718).

PERFORMANCE LIMITATIONS OF BLOCK–MULTITHREADED DISTRIBUTED–MEMORY SYSTEMS

W.M. Zuberek

Department of Computer Science
Memorial University
St.John's, Canada A1B 3X5

and

Department of Applied Informatics
University of Life Sciences
02-787 Warsaw, Poland

ABSTRACT

The performance of modern computer systems is increasingly often limited by long latencies of accesses to the memory subsystems. Instruction–level multithreading is an architectural approach to tolerating such long latencies by switching instruction threads rather than waiting for the completion of memory operations. The paper studies performance limitations in distributed–memory block multithreaded systems and determines conditions for such systems to be balanced. Event–driven simulation of a timed Petri net model of a simple distributed–memory system confirms the derived performance results.

1 INTRODUCTION

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, until recent years the performance of processors has been doubling every 18 months (the so–called Moore's law (Hamilton 1999)). However, the performance of memory chips has been improving by only 10% per year (Rixner et al. 2000), creating a "performance gap" in matching processor's performance with the required memory bandwidth. Detailed studies have shown that the number of processor cycles required to access main memory has been doubling approximately every six years (Sinharoy 1997). 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 and it is not unusual that as much as 60% of processor's time is spent on waiting for the completion of memory operations (Sinharoy 1997).

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 (Chen and Bauer 1994, Klaiber and Levy 1991), speculative loads and execution (Rogers at al. 1992) and multithreading (Agarwal 1992; Byrd and Holliday 1995; Ungerer et al. 2003, Chaudhry et al. 2005, Emer et al. 2007).

Instruction–level multithreading, and in particular block–multithreading (Agarwal 1992; Boothe and Ranada 1992; Byrd and Holliday 1995), tolerates long–latency memory accesses and synchronization delays by switching the threads rather than waiting for the completion of a long–latency operation which can require hundreds or even thousands of processor cycles (Emer at al. 2007). It is believed that the return on multithreading is among the highest in computer microarchitectural techniques (Chaudhry at al. 2005).

In distributed–memory systems, the latency of memory accesses is even more pronounced than in centralized-memory systems because memory access requests 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, typically assigned to the switches that control the traffic between the nodes of the interconnecting network (Govindarajan at al. 1995).

The mismatch of performance among different components of a computer system significantly impacts the overall performance. If different components of a system are utilized at significantly different levels, the component which is utilized most intensively will first reach its limit (i.e., utilization close to 100%), and will restrict the utilization of all other elements as well as the performance of the whole system; such an element is called a bottleneck. A system which contains a bottleneck is unbalanced. In balanced systems, the utilizations of all components are (approximately) equal, so the performance of the system is maximized because all system components reach their performance limits at the same time.

The purpose of this paper is to study performance limitations in distributed–memory block multithreaded systems by comparing service demands for different components of the system. The derived results are confirmed by event–driven simulation of a timed Petri net instruction–level model of the analyzed system.

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 Figure 1.

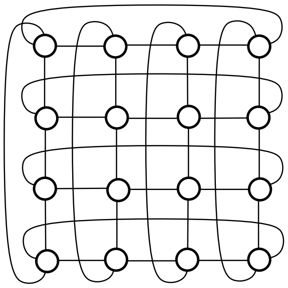

Figure 1: Outline of a 16–processor system.

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 (non-uniform distribution of memory accesses, representing a sort of spacial locality of memory accesses, can be taken into account by adjusting the average number of hops).

Although many specific details refer to this 16–processor system, most of them can easily be adjusted to other systems by changing the values of a few model parameters.

Each node in the network shown in Figure 1 is a block multithreaded processor which contains a processor, local memory, and two network interfaces, as shown in Figure 2. 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.

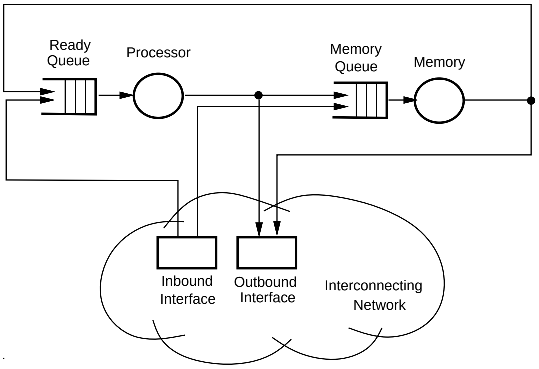

Figure 2: Outline of a single multithreaded processor.

Figure 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 (local, 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 Figure 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 and improving their performance is the key to the improved performance of the entire system (Jain 1991).

2 TIMED PETRI NET MODEL

Petri nets have become a popular formalism for modeling systems that exhibit parallel and concurrent activities (Reisig 1985, Murata 1989). In timed nets (Zuberek 1991, Wang 1998), deterministic or stochastic (exponentially distributed) firing times are associated with transitions, and transition firings are timed events, 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 instructions execution is shown in Figure 3. 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. 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.

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

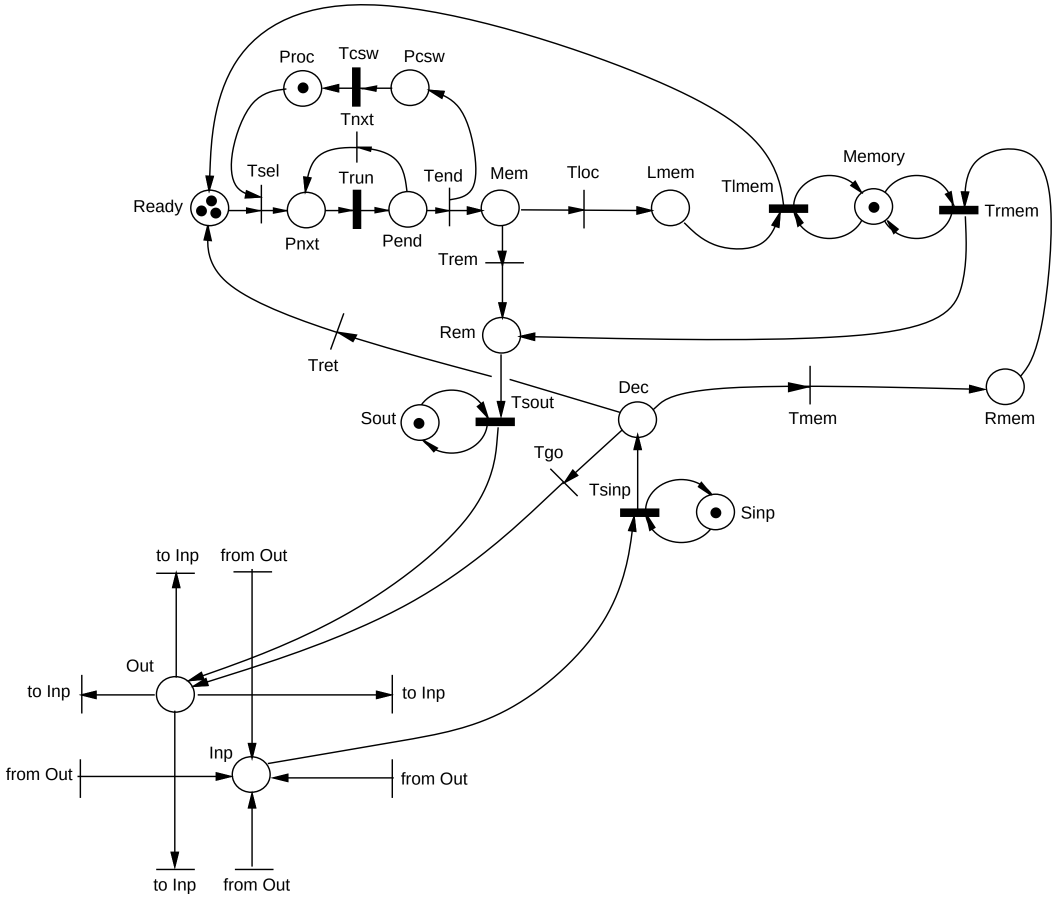

Figure 3: Instruction–level Petri net model of a block multithreaded processor.

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 (Govindarajan 1997). For a 16–processor system (as in Figure 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*).

It can be observed that the remote memory access requests do not guarantee that the results of memory accesses return to the requesting (’home’) nodes. Although a more detailed model representing the routing of messages can be developed using colored Petri nets (Zuberek at al. 1998), such a more detailed model provides results which are insignificantly different from a simpler model, discussed in this section. Consequently, only this simpler model is used in performance analysis that follows.

3 PERFORMANCE ANALYSIS

The parameters which characterize the model of the block multithreaded distributed–memory system include:

n_p – the number of processors,
n_t – the number of threads,
ℓ_t – the thread runlength,
t_p – the processor cycle time,
t_m – 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, $p_r = 1 - p_\ell$.

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 assumed 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.

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 at the processor is simply the thread runlength, ℓ_t.

The service demand at 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 $p_r * t_m$.

The service demand due to a single thread (in each processor) at the inbound switch is 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 in the multiprocessor system. Hence, the service demand due to a single thread in an inbound switch is $2 * p_r * n_h * t_s$. For the outbound switch, the service demand is $d_{so} = 2 * t_s * p_r$; the number of hops, n_h, does not affect d_{so} since each remote request and its response go through the outbound switch once at the source and once at the destination processor (this also explains the factor 2 in the above formula).

The service demands are thus:

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

A balanced system is usually defined as a system in which the service demands for all components are equal (Jain 1991). So, in a balanced system, $\ell_t = t_m = 2 * p_r * n_h * t_s$ (since d_{so} is always smaller than d_{si} for $n_h \geq 1$, the output switch cannot become the system's bottleneck and is therefore disregarded in balancing the system; in the discussed system this switch is simply "underutilized").

Figure 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.

Since for a 16–processor system $n_h \approx 2$ (Zuberek 2000), so for $t_s = 10$ the balance is obtained for $p_r = 0.25$ (or $p_\ell = 0.75$), which is very clearly demonstrated in Figure 4 as the "edge" of the high utilization region. Figure 5 shows the utilization of the processor and the switch as functions of p_r, the probability of accessing remote memory (the processor utilization plot corresponds to the crossection of Figure 4 at $n_t = 10$). It can be observed that the only region of high utilization of the processor is when the switch is utilized less that 100% (and then it is not the bottleneck). The balance corresponds to the intersection point of the two plots.

If the information is uniformly distributed among the nodes of the distributed–memory system, the value of $p_r = (n_p - 1)/n_p$, so the utilization of processors is rather low in this case (close to 0.3 in Figure 4). This indicates that the switches are simply too slow for this system.

There are two basic ways to reduce the limiting effects of the switches; one is to use switches with smaller switch delay (for example, $t_s = 5$), and the other is to use parallel switches and to distribute the workload among them. It appears that both solutions produce very similar results (Zuberek 2002) because what counts in the case of heavily used components is the throughput rather than the switch delay.

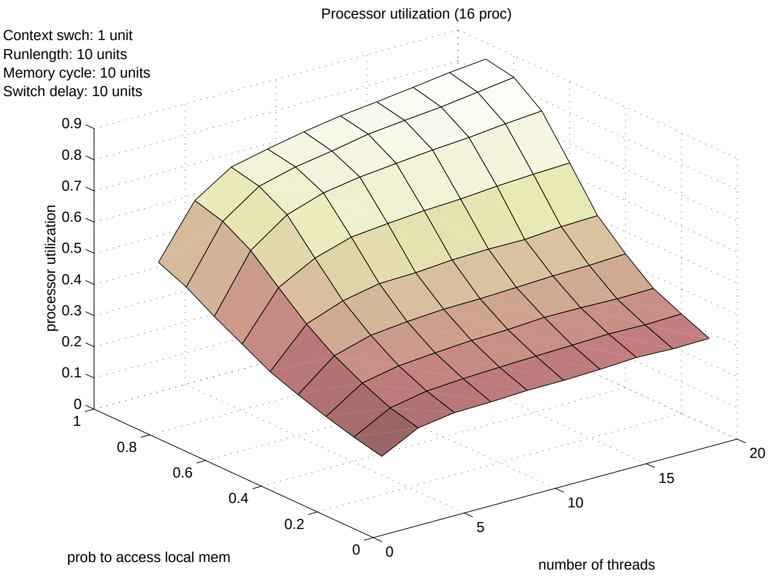

Figure 4: Processor utilization – 16 processor system;
$t_{cs} = 1$, $\ell_t = 10$, $t_m = 10$, $t_s = 10$.

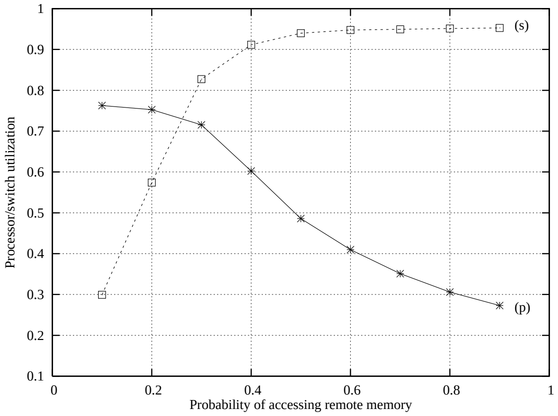

Figure 5: Utilization of processors (p) and switches (s) – 16 processor system;
$t_{cs} = 1$, $\ell_t = 10$, $t_m = 10$, $t_s = 10$, $n_t = 10$.

Figure 6 shows the utilization of the processors in the same 16–processor system when the switch delay is reduced to 5 processor cycles.

It can be observed that the region of high utilization of the processors is significantly extended, but there is still the limiting effect of the switches for p_r close to 1 (or p_ℓ close to 0). Figure 7 shows the balance for the case of $t_s = 5$ and $n_t = 10$ in greater detail.

The balance is now obtained for $p_r = 0.5$, which is still quite distant from the values corresponding to the uniform distribution of accesses among the nodes of the system.

A reorganization of the information in the distributed memory may be possible to make as much information needed for processing local as possible. If such a reorganization can reduce the probability p_r from almost 1 to say 0.5, the balance could be obtained without further hardware improvements.

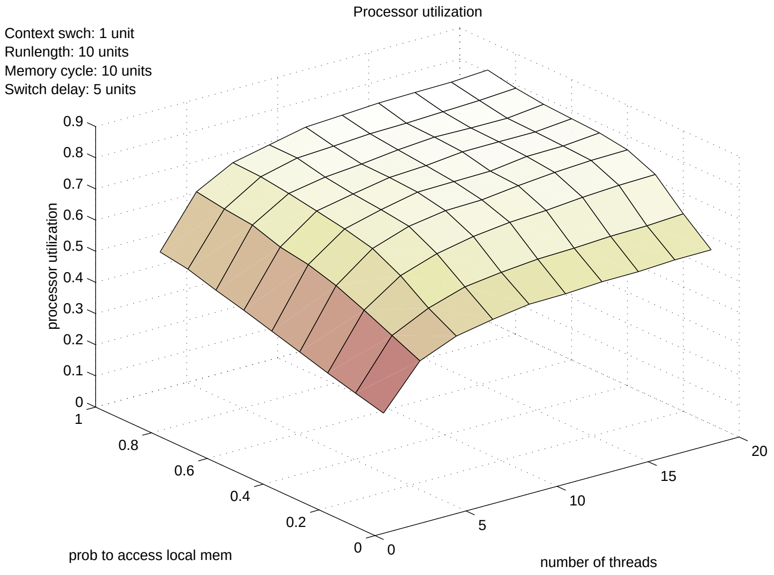

Figure 6: Processor utilization – 16 processor system;
$t_{cs} = 1$, $\ell_t = 10$, $t_m = 10$, $t_s = 5$.

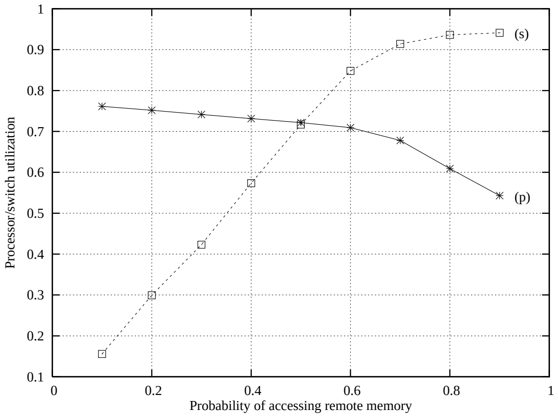

Figure 7: Utilization of processors (p) and switches (s) – 16 processor system;
$t_{cs} = 1$, $\ell_t = 10$, $t_m = 10$, $t_s = 5$, $n_t = 10$.

4 CONCLUDING REMARKS

The paper presents a timed Petri net model of block multiprocessor system at the instruction execution level, and analyzes the effects of system bottlenecks on the performance of system components. 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 practically doubles the utilization of processors in the critical region of small values of p_ℓ (Figures 4 and 6).

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; simulation studies (Zuberek 2002) 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 keep this utilization high. The simplest way to achieve this is to make the processor the bottleneck of the system.

The results obtained for a 2–dimensional torus–like network are also valid for other interconnecting networks with the same connectivity characteristics. For example, Figure 8 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 Figure 1, the performance characteristics of both networks are also the same (although the two interconnecting networks scale in different ways).

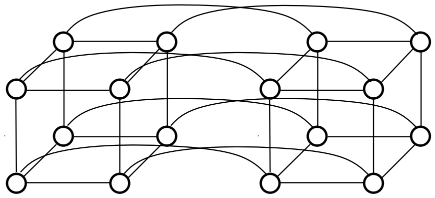

Figure 8: Outline of a 16–processor system.

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 *Dec* (and, consequently, the value of n_h).

ACKNOWLEDGEMENT

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

REFERENCES

Agarwal, A. 1992. Performance tradeoffs in multithreaded processors; *IEEE Transactions on Parallel and Distributed Systems* 3 (5): 525-539.

Boothe, B. and A. Ranade. 1992. Improved multithreading techniques for hiding communication latency in multiprocessors. In *Proceedings of the 19-th Annual International Symposium on Computer Architecture*, 214-223.

Byrd, G.T. and M.A. Holliday. 1995. Multithreaded processor architecture. *IEEE Spectrum* 32 (8): 38-46.

Chen, T-F. and J-L. Baer. 1994. A performance study of software and hardware data prefetching scheme. In *Proceedings of the 21-st Annual International Symposium on Computer Architecture* 223-232.

Chaudhry, S., P. Caprioli, S. Yip and M. Tremblay. 2005. High–performance throughput computing. *IEEE Micro* 25 (3): 32-45.

Emer, J., M.D. Hill, Y.N. Patt, J.J. Yi, D. Chiou and R. Sendag. 2007. Single-threaded vs. multithreaded: where should we focus? *IEEE Micro* 27 (6): 14-24.

Govindarajan, R., S.S. Nemawarkar and P. LeNir. 1995. Design and performance evaluation of a multithreaded architecture. In *Proceedings of the First IEEE Symposium on High–Performance Computer Architecture* 298-307.

Govindarajan, R., F. Suciua and W.M. Zuberek. 1997. Timed Petri net models of multithreaded multiprocessor architectures. In *Proceedings of the 7-th International Workshop on Petri Nets and Performance Models* 153-162.

Hamilton, S. 1999. Taking Moore's law into the next century. *IEEE Computer Magazine* 32 (1): 43-48.

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

Klaiber, A.C. and H.M. Levy. 1991. An architecture for software-controlled data prefetching. In *Proceedings of the 18-th Annual International Symposium on Computer Architecture* 43-53.

Murata, T. 1989. Petri nets: properties, analysis and applications. *Proceedings of IEEE* 77 (4): 541-580.

Reisig, W. 1995. *Petri nets – an introduction* (EATCS Monographs on Theoretical Computer Science 4). Springer–Verlag.

Rixner, S., W.J. Dally, U.J. Kapasi, P. Mattson and J.D. Ovens. 2000. Memory access scheduling. In *Proceedings of the 27-th Annual International Symposium on Computer Architecture* 128-138.

Rogers, A. and K. Li. 1992. Software support for speculative loads. In *Proceedings of the 5-th Symposium on Architectural Support for Programming Languages and Operating Systems* 38-50.

Sinharoy B. 1997. Optimized thread creation for processor multithreading. *The Computer Journal* 40 (6): 388-400.

Ungerer, T., B. Robic and J. Silc. 2003. A survey of processors with explicit multithreading. *ACM Computing Surveys* 35 (1): 29-63.

Wang, J. 1998. *Timed Petri nets.* Kluwer Academic Publ.

Zuberek, W.M. 1991. Timed Petri nets – definitions, properties and applications. *Microelectronics and Reliability* 31 (4): 627-644.

Zuberek, W.M. 2000. Performance modeling of multithreaded distributed memory architectures. In *Hardware Design and Petri Nets* 311-331. Kluwer Academic Publ.

Zuberek, W.M. 2002. Analysis of performance bottlenecks in multithreaded multiprocessor systems. *Fundamenta Informaticae* 50 (2): 223-241.

Zuberek, W.M., R. Govindarajan and F. Suciu. 1998. Timed colored Petri net models of distributed memory multithreaded multiprocessors. In *Proceedings of the Workshop on Practical Use of Colored Petri Nets and Design/CPN* 253-270.

AUTHOR BIOGRAPHY

WLODEK M. ZUBEREK received M.Sc. degree in Electronic Engineering and Ph.D. and D.Sc. degrees in Computer Science, all from Warsaw Technical University. Currently he is a Professor in the Department of Computer Science of Memorial University in St.John's, Canada, and is also associated with the Department of Applied Informatics of the University of Life Sciences in Warsaw, Poland. 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 email address is <wlodek@mun.ca>.