2-nd Int. Conference on Application of Concurrency to System Design, Newcastle, UK, June 25–29, 2001, pp.43-52.
Copyright © IEEE (DOI 10.1109/CSD.2001.981763).

Analysis of Performance Limitations in Multithreaded Multiprocessor Architectures

W.M. Zuberek
Department of Computer Science
Memorial University of Nfld
St.John's, Canada A1B 3X5
wlodek@cs.mun.ca

Abstract

The performance of modern multiprocessor systems is increasingly limited by interconnection delays or long latencies of memory subsystems. Instruction–level multithreading is a technique to tolerate such long latencies by switching from one instruction thread to another and continuing instruction execution concurrently with the long–latency operations. Using timed Petri net models, the paper analyzes performance limitations introduces by different components of distributed–memory multithreaded multiprocessor systems. Simulation results are used to compare performance improvements obtained by replicating critical components of the system to those obtained using components with better performance characteristics.

1. Introduction

Modern multiprocessor systems are becoming increasingly limited by the performance of components other than the processor, mainly memory subsystems and interconnections. Due to continuous progress in fabrication technologies, the performance of processors have been doubling every 18 months (the so–called Moore's law [9]). However, the bandwidth of memory chips has been increasing by only 10% per year [16], which makes it difficult to provide the memory bandwidth required to match the processor performance. Also, the increasing density of on–chip circuits reduces the distances the electrical signals must traverse between consecutive operations, but it also increases the difference of communication delays between on–chip and off–chip operations. In effect, it is becoming increasingly the case that the performance of applications depends on the performance of components other than the processor.

Much research has focused on reducing and tolerating memory access latencies. Techniques for reducing the frequency and impact of cache misses include hardware and software prefetching [5, 12], speculative loads and speculative execution [17] and multithreading [1, 4].

As effective memory latency is reduced, memory bandwidth consumption increases. For example, when processor simultaneously fetches k data items from memory, the effective latency per data item is reduced k times, but the memory bandwidth consumption increases k times. Since the effective latency is the inverse of the consumed bandwidth [6], memory latency cannot be fully tolerated without infinite bandwidth. This is the reason that in many real machines, program performance is bounded by the limited rate at which data operands are delivered to the processor, regardless of the processor's speed. Comprehensive analysis of bandwidth, that is required at different stages of memory hierarchy for program execution, is presented in [6]; several program transformations which improve program performance by reducing its memory bandwidth requirements are also proposed in [6].

To maximize the memory bandwidth, modern DRAM components allow pipelining of memory accesses, provide several independent memory banks, and cache the most recently accessed row of each bank [16]. While these features increase the peak supplied memory bandwidth, they also make the performance of the DRAM highly dependent on the access pattern.

In distributed–memory systems, the effects of long–latency operations are even more pronounced as the memory access requests must often perform a number of hops from one node of the distributed system to another to reach its destination and then the results must be transferred back to the originating node. The delays (and congestion) in the interconnecting network is often another factor limiting the performance of the multiprocessor systems.

Instruction–level multithreading, and in particular block–multithreading [1, 2, 4], tolerates long–latency memory accesses and synchronization delays by switching to another thread rather than waiting for the completion of a long–latency operation which, in a distributed–memory system, can require hundreds or even thousands of processor cycles. A combination of multithreading and superscalar architecture is also an approach used in high–performance microprocessors [13].

The growing mismatch between the performances of processors and their memories results in memory being often the system's bottleneck, and limiting the performance of all other components of the system. In distributed–memory systems, the interconnecting network can be another source of performance limitations. The purpose of this paper is to study the performance of distributed–memory multithreading systems, and in particular, the performance limitations introduced by memory and the interconnecting network. Since simply reducing the memory latency or the delays of interconnecting networks may not be possible, the paper studies the effects of additional concurrency introduced by replicating the critical elements of the system. In the case of memory, this corresponds to splitting the memory into thread–level independent banks. For the interconnecting network, the performance can be improved by using multiple parallel switches. Timed Petri nets [19] are used to model several multithreaded multiprocessor systems at the instruction execution level, and simulation of these models provides performance characteristics of the analyzed multithreaded systems.

2. Multithreaded multiprocessor systems

A multiprocessor 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.1.1.

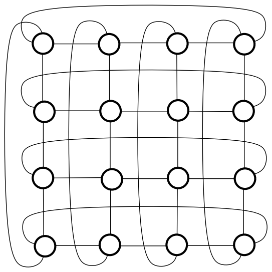

Fig.1.1. Outline of a 16–processor system.

It is usually assumed that the 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, is usually determined assuming that the memory accesses are uniformly distributed over the nodes of the system. This average length, denoted by n_h, is one of modeling parameters (for a 16–processor system, with a uniform distribution of accesses over the nodes, the value of n_h is close to 2 [8]; in general, for a system with $p \times p$ processors connected by a 2–dimensional torus network, n_h can be approximated reasonably well by $p/2$).

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 only a few parameters.

Each node in the system shown in Fig.1.1 is a multithreaded processor which contains a processor, local memory, and two network interfaces, as shown in Fig.1.2. 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.

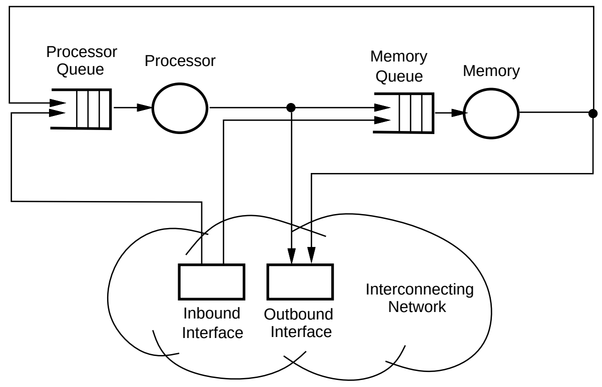

Fig.1.2. Outline of a single multithreaded processor.

Fig.1.2 also shows a queue of ready threads (Processor Queue); whenever the processor performs a context switching (i.e., switches from one thread to another), a thread is selected from this queue 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, and is one of important modeling parame-

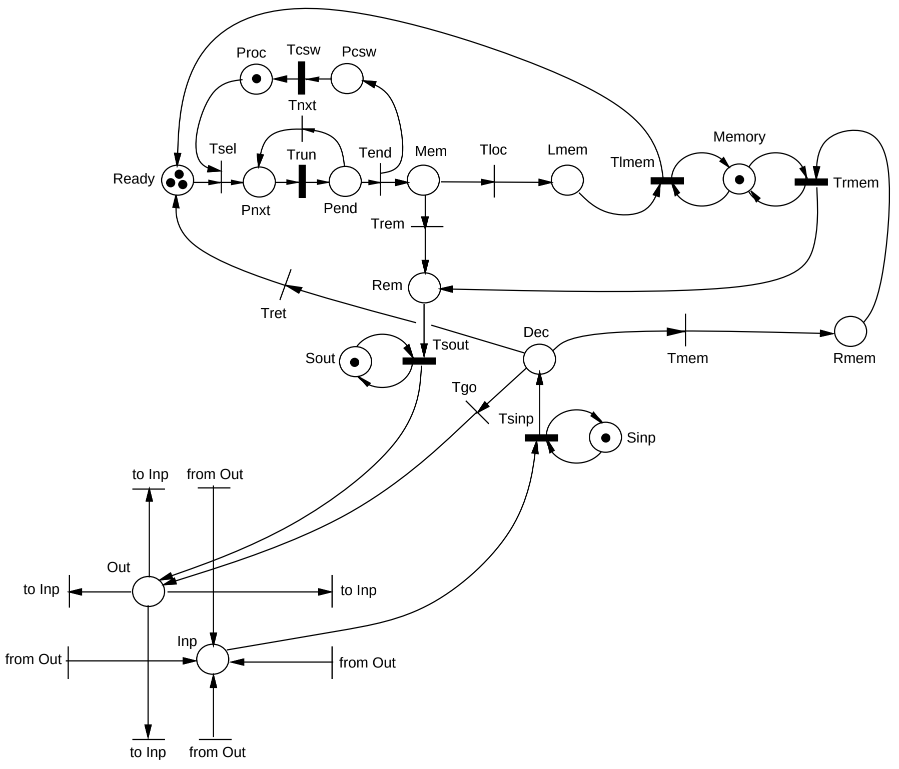

Fig.2.1. Instruction–level Petri net model of a multithreaded processor.

ters. 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.1.2 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.

In Fig.1.2, the memory is represented as a single "server" with the service time corresponding to the average time of performing memory operations. In real systems, the memory system is hierarchical and contains several levels of memories with different performance characteristics (e.g., assuming that the first level cache is integrated with the processor, the memory system may include the second level cache, possibly the third level cache, and main memory). It appears, however, that the detailed representation of memory hierarchy has only minor effect on the performance characteristics of the system, so a simple model (as shown in Fig.1.2) with the average characteristics is often satisfactory.

The (average) number of available threads, n_t, is yet another modeling parameter. In order to simplify the models, it is assumed that the value of n_t does not change during program execution.

3. Petri net models

Petri nets [15, 14] are popular models of systems that exhibit concurrent and parallel activities. In timed Petri nets [19], the durations of modeled activities are also taken into account in order to study the performance characteristics of the systems.

A timed Petri net model of a multithreaded processor at the level of instruction execution is shown in Fig.2.1 [20]. 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

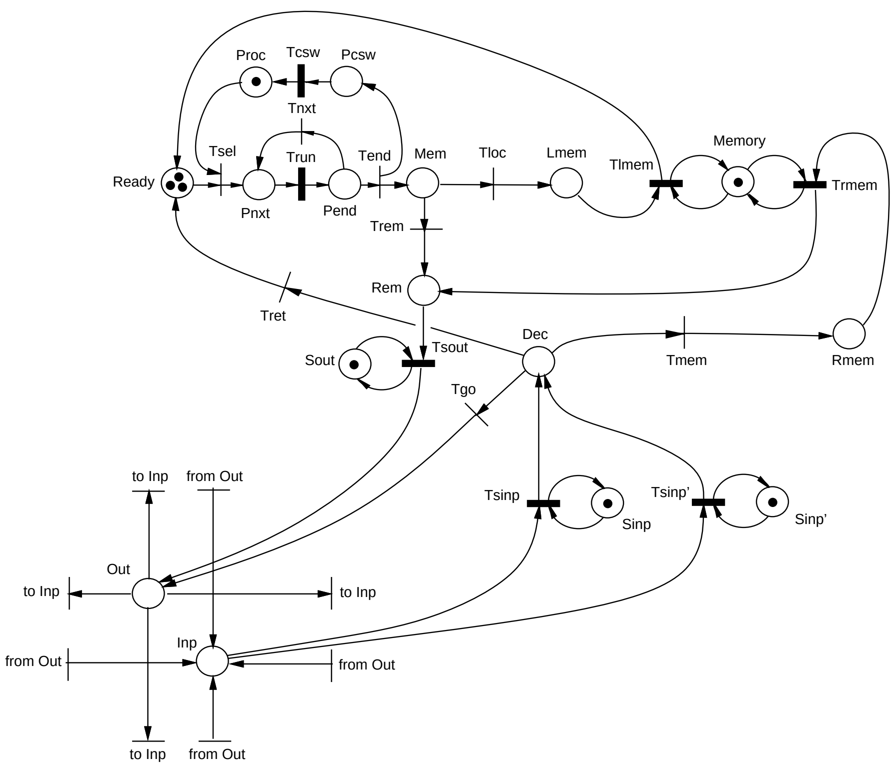

Fig.2.2. Petri net model of a multithreaded processor with two inbound switches.

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 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 [8]. For a 16–processor system (as in Fig.1.1), 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,

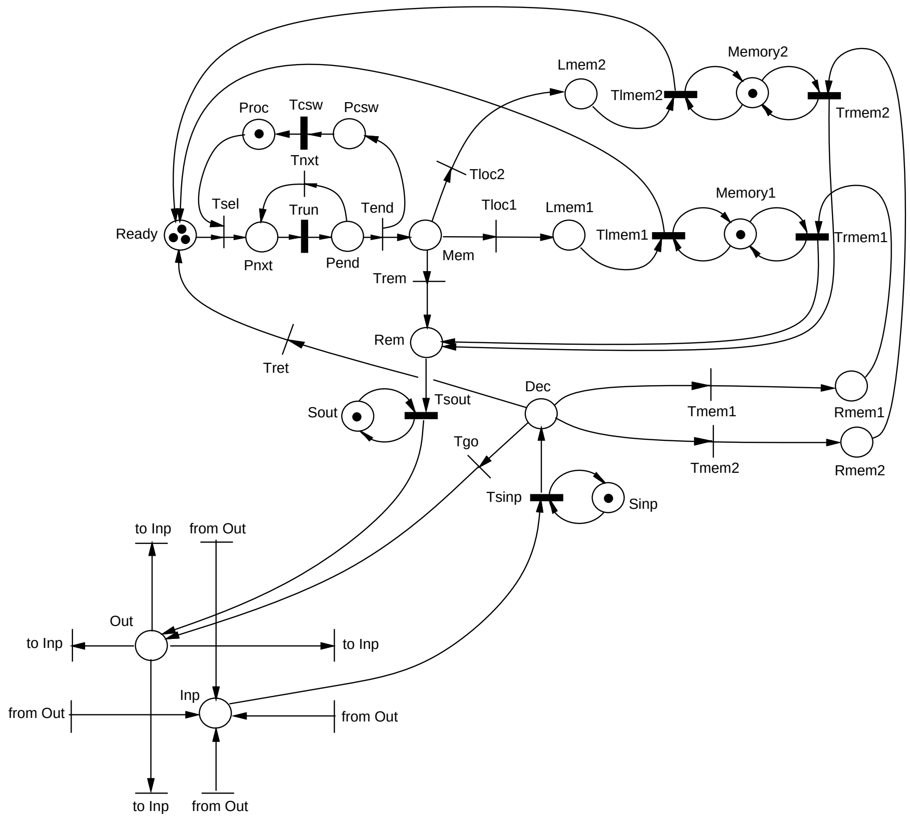

Fig.2.3. Petri net model of a multithreaded processor with two memory banks.

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 $Trem$).

If the performance of the system is limited by the switches in the interconnecting network (i.e., if the switch is the bottleneck in this systems), the two most obvious remedies are (i) to reduce the delay introduced by the switch (i.e., the firing time associated with transitions $Tsinp$ and $Tsout$), or (ii) to introduce multiple switches which does not reduce the latency, but which increases the throughput of the interconnecting network, so it reduces the times spent on waiting in the queues, and increases the performance of the system.

A Petri net model of a processor with two inbound switches is shown in Fig.2.2. The delay of each switch is the same, t_s, but if there is more than one request to be forwarded to other nodes, both switches will handle the traffic at the same time, increasing the throughput of the interconnecting network and reducing the delays imposed by the network.

It should be observed that a model equivalent to that in Fig.2.2 can be obtained from the one shown in Fig.2.1 by increasing the initial marking of place $Sinp$ to the number of switches, in this case two.

In the case when the memory is the component limiting the performance of the system, and simply reducing the access time to memory is not an available option, the performance can be increased by splitting the memory into two (or more) banks in such a way that all banks are accessed with the same probability. It should be noticed that such a splitting is introduced exclusively for multithreading and should not be confused with memory interleaving and other techniques used to increase the throughput of memory at the instruction–execution level. Consequently, this thread–level splitting does not affect the memory access time, so the banks retain their original performance characteristics.

A Petri net model of a system with two memory banks is shown in Fig.2.3, in which the representation of memory (place $Memory$) is replicated with its all adjacent transitions, and connected with the remaining part of the model through free–choice places Mem and Dec. In effect, all local memory accesses have now additional choice of selecting the first or the second memory bank (immediate transitions $Tloc1$ and $Tloc2$); similarly, remote memory access requests, after reaching the target node, must select one of the

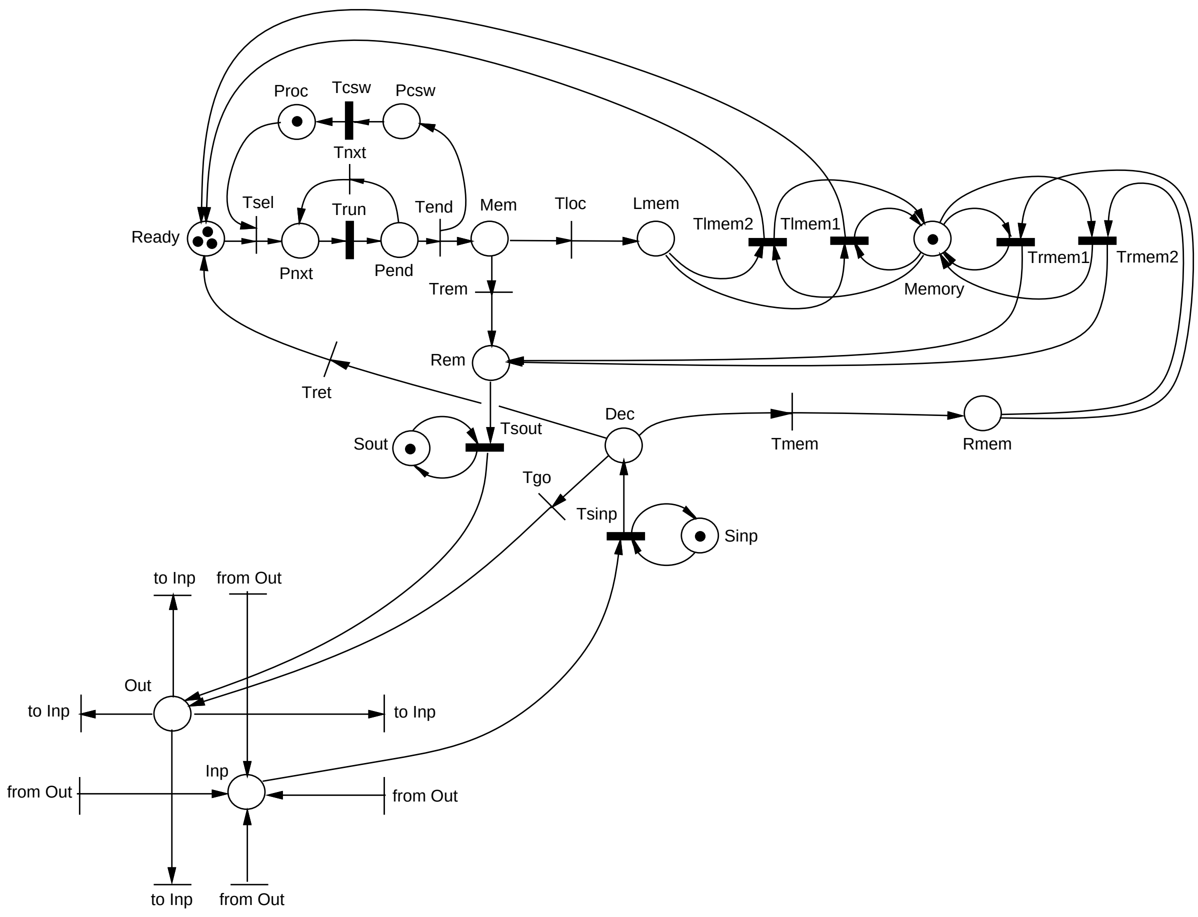

Fig.2.4. Petri net model of a multithreaded processor with two levels of memory.

two memory banks (immediate transitions $Tmem1$ and $Tmem2$). It is assumed that the accesses are uniformly distributed, so the selection of each memory bank is equally probable.

It should be noticed that the model shown in Fig.2.3 cannot be equivalently represented by the model shown in Fig.2.1 with two initial tokens in place *Memory*; the difference is due to the fact that in case of memory, several requests can be queued to the same bank while the other bank is idle.

Fig.2.4 shows another refinement of the model shown in Fig.2.1, which introduces two levels of memory (e.g., second level cache and main memory). These two levels of memory, with significantly different performance characteristics, are represented as additional choices for (local and remote) accesses; for example, *Tlmem1* and *Trmem1* can represent accesses to second level cache and *Tlmem2* and *Trmem2* accesses to main memory. The firing times associated with these transitions and their choice probabilities will typically differ by an order of magnitude.

Finally, in cases when the processor is the bottleneck, a replication of the processor (or some of its parts) should be considered. For example, in "simultaneous multithreading" [7, 18] several threads simultaneously issue instructions to a common set of execution units.

4. Performance results

It is convenient to assume that all timing characteristics are expressed in processor cycles (which is assumed to be 1 unit of time). The basic model parameters and their typical values are as follows:

symbol	*parameter*	*values*
n_t	the (average) number of threads	2,...,20
ℓ_t	thread runlength	5,10,20
t_{cs}	context switching time	1,2,5
t_m	memory cycle time	10,20
t_s	switch delay	5,10
p_ℓ, p_r	probability of accesses to local/remote memory	0.1,...,0.9

Fig.3.1 shows the utilization of the processor 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 modeling parameters.

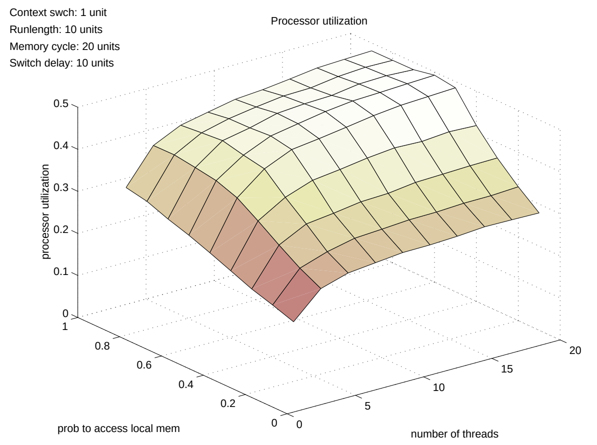

Fig.3.1. Processor utilization; $t_{cs} = 1$, $\ell_t = 10$, $t_m = 20$, $t_s = 10$.

It can be observed that, for values of p_ℓ close to 1 (i.e., when most of memory references are to local memory), the utilization increases with the number of available threads, n_t, and tends to the bound 0.5 which is determined, in this case, by the ratio of ℓ_t/t_m (the memory is the bottleneck in this region).

For smaller values of p_ℓ, the utilization of the processor "saturates" very quickly and is practically insensitive to the number of available threads n_t. This is a clear indication that some other component of the system is the bottleneck (i.e., a component with utilization close to 100%, which limits the performance of all other components of the system).

The bottlenecks can be identified by comparing service demands for the different components of the system [10]; the component with the highest service demand is the first one to reach its utilization bound (i.e., utilization of almost 100%), so it is the bottleneck that limits the utilization of all other components of the system.

The service demands (per one long–latency memory access) are [20]:

component	*service demand*
processor	ℓ_t
memory	t_m
inbound switch	$2 * p_r * n_h * t_s$
outbound switch	$2 * p_r * t_s$

If $t_m = 2\ell_t$, $\ell_t = t_s$, and $n_h = 2$ (as in Fig.3.1), the inbound switch becomes the bottleneck for $p_r > 0.5$ (or $p_\ell < 0.5$); for $p_r < 0.5$ (or $p_\ell > 0.5$) the memory is the bottleneck. The utilization of memory is shown

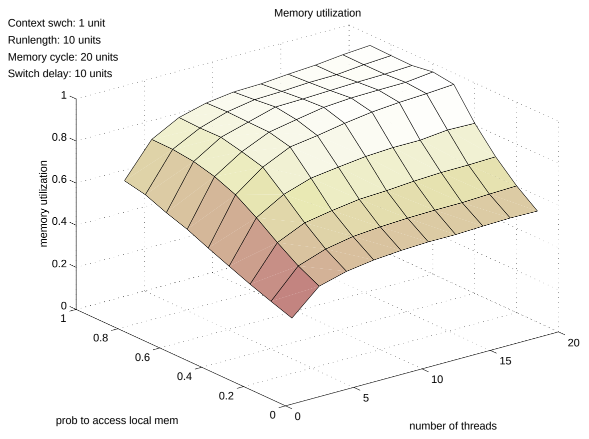

Fig.3.2. Memory utilization; $t_{cs} = 1$, $\ell_t = 10$, $t_m = 20$, $t_s = 10$.

in Fig.3.2, and it can be easily observed that Fig.3.2 is very similar to the utilization of the processor (Fig.3.1) but at the level close to 100%, i.e., twice the utilization of the processor (as the consequence of $t_m = 2\ell_t$).

Fig.3.3 shows the utilization of the processor (also as a function of the number of available threads and the probability of long–latency accesses to local memory) for the case of memory composed of two banks, with equal probabilities of accessing each of them. For values of p_ℓ close to 1, the utilization of the processor in Fig.3.3 tends to its limiting value which, due to the overhead of context switching, is equal to $\ell_t/(\ell_t + t_{cs}) = 0.91$.

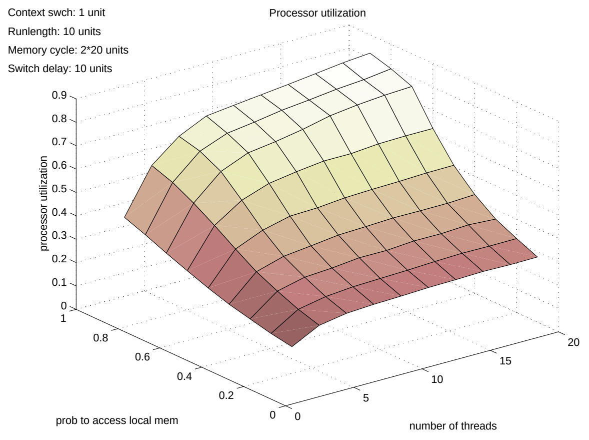

Fig.3.3. Processor utilization; $t_{cs} = 1$, $\ell_t = 10$, $t_m = 2 \parallel 20$, $t_s = 10$.

The effects of splitting the memory in two independent banks are practically the same as using a single memory bank with one half of the original latency; Fig.3.4 shows the utilization of the processor for the case when the memory cycle time is reduced from 20 to 10 processor cycles. The results in Fig.3.4 differ insignificantly from those in Fig.3.3.

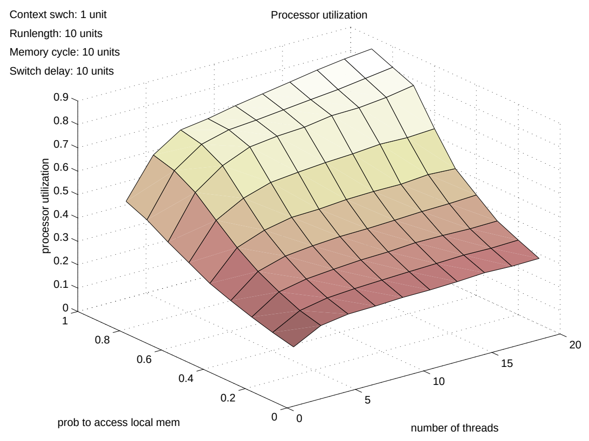

Fig.3.4. Processor utilization; $t_{cs} = 1$, $\ell_t = 10$, $t_m = 10$, $t_s = 10$.

It should be also observed that the processor's utilization is sensitive to memory latency only in the region in which the memory is the bottleneck; for small values of p_ℓ (or p_r close to 1), the utilization is practically the same in all cases (Fig.3.1, Fig.3.3 and Fig.3.4).

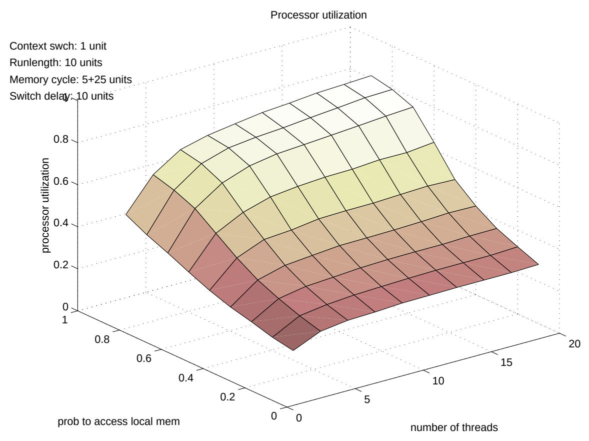

Fig.3.5. Processor utilization; $t_{cs} = 1$, $\ell_t = 10$, $t_{m1} = 5$, $t_{m2} = 25$, $t_s = 10$.

Fig.3.5 shows the utilization of the processor for the case when several levels of memory are explicitly represented in the model. Two levels of memory are taken into account in this case, as shown in Fig.2.4. The parameters of memory levels are selected in such a way that the average memory access time is consistent with that in Fig.3.4 (i.e., one level of memory has access time of 5 time units and choice probability of 0.75, and the other level has access time of 25 time units, and the choice probability of 0.25). The results shown in Fig.3.5 are practically the same as in Fig.3.4 and Fig.3.3, which is another indication that a detailed model of the memory system is needed only for very accurate performance analyses.

For small values of p_ℓ, the utilization of the processors shown in Fig.3.3 and Fig.3.4 is rather low which indicates that some other component of the system becomes the bottleneck in this region. Indeed, a comparison of service demands shows that it is the input switch, which, for $p_\ell < 0.75$, becomes the bottleneck. Fig.3.6 shows the utilization of the input switch as a function of the number of available threads, n_t, and the probability p_r of accessing remote (not local) memory, so the front part of Fig.3.6 corresponds to the back part of Fig.3.3.

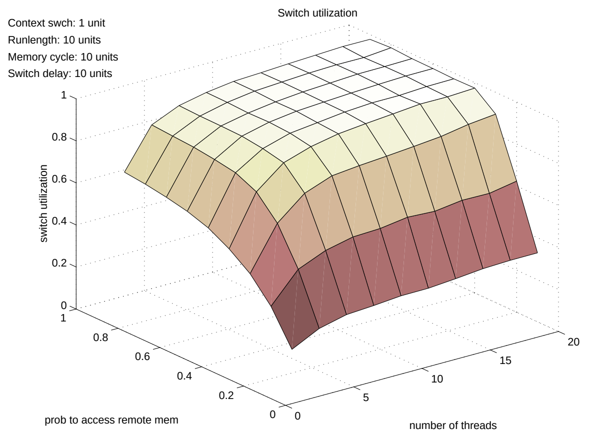

Fig.3.6. Switch utilization; $t_{cs} = 1$, $\ell_t = 10$, $t_m = 10$, $t_s = 10$.

Fig.3.6 clearly shows that with the exception of small values of n_t or p_r, the input switch is utilized practically in 100%, which means that it is simply "too slow" for this system.

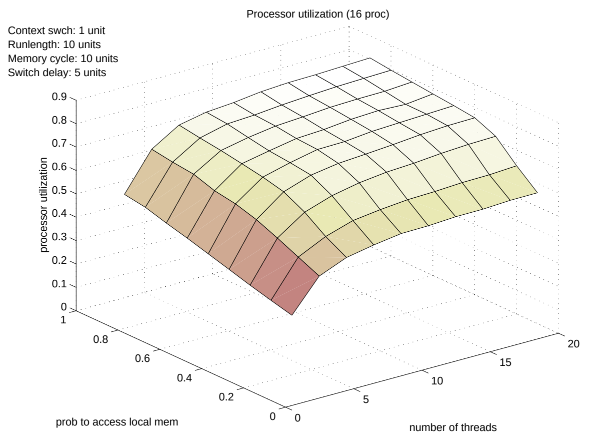

Fig.3.7. Processor utilization; $t_{cs} = 1$, $\ell_t = 10$, $t_m = 10$, $t_s = 5$.

Fig.3.7 shows the utilization of the processor for the case when the switch delay is reduced from 10 to 5 processor cycles. The extended region of high utilization of the processor, due to the reduced switch delay, can be easily observed in Fig.3.7.

The utilization of the processor with two parallel switches (as shown in Fig.2.2) used instead of a single switch with reduced delay, is shown in Fig.3.8. These results are practically the same as in Fig.3.7.

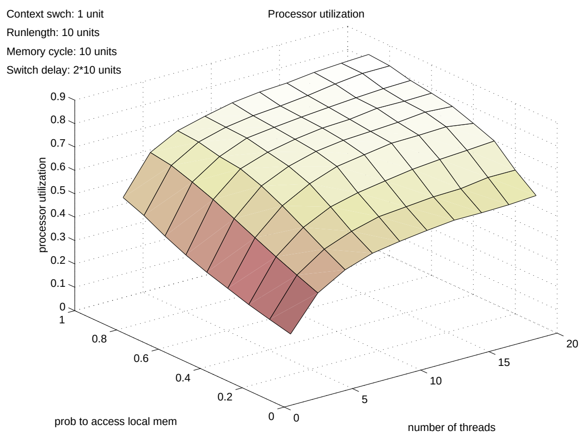

Fig.3.8. Processor utilization; $t_{cs} = 1$, $\ell_t = 10$, $t_m = 10$, $t_s = 2 \parallel 10$.

5. Concluding remarks

The presented performance analysis of multithreaded multiprocessor systems shows that performance improvements that can be achieved by using components with improved performance characteristics are practically the same as obtained by replicating the critical components of the system. This may be especially attractive when the components with improved performance characteristics are not available; the replication of critical components can still provide significant improvements of the performance of the whole systems. Such improvements can also be used to balance the system, i.e., to utilize all components of the system at (approximately) the same level.

The utilization of processors with several replicated components is shown in Fig.4.1. In this case, there are two parallel switches (as in Fig.2.2), each with delay $t_s = 10$ time units, and two banks of memory (as in Fig.2.3), each with two levels (as in Fig.2.4) characterized by access times $t_{m1} = 10$ time units with probability 0.75, and $t_{m2} = 50$ time units with probability 0.25. Consequently, the maximum throughput of the memory component (when memory is the bottleneck) is 0.1 accesses per time unit which corresponds to the effective access time of 10 time units. Similarly, the effective switch delay (when switch is the bottleneck) is 5 time units.

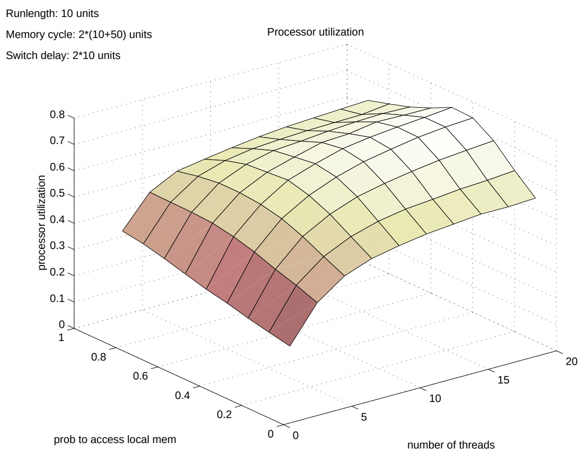

Fig.4.1. Processor utilization; $t_{cs} = 1$, $\ell_t = 10$, $t_m = 2 \parallel (10 * 0.75 + 50 * 0.25)$, $t_s = 2 \parallel 10$.

Comparing Fig.4.1 with Fig.3.7 it can be observed that significant differences are only in the region in which the values of p_ℓ are close to 1, in which the memories and the switches are not utilized fully. In this region the performance depends on the delays introduced by individual components rather than the (total) throughputs; the results shown in Fig.4.1 represent gradual transition from the results shown in Fig.3.1 to those shown in Fig.3.7.

The results presented in this paper indicate that only a small number of threads is needed to achieve the performance close to its upper bound; the influence of additional threads beyond 6 to 8 is rather insignificant in all presented cases.

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.

The developed performance models can be used for approximate characterization of the performance [21]. 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 performance of the whole system. Identification of this limiting component (called the bottleneck) also allows to estimate the performance of the system.

Finally, it can be observed that Petri net models of multiprocessor systems contain many "regularities" which can be used for model reduction. For example, in colored Petri nets [11], tokens are associated with attributes (called colors), so different activities can be associated with tokens of different types. An immediate application of colors is to represent the different processors (or nodes) by different colors within the same processor model; consequently, a colored Petri net will need only one processor model (for any number of processors). Some other aspects of colored net models are discussed in [8].

Acknowledgments

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., Holliday, M.A., "Multithreaded processor architecture"; IEEE Spectrum, vol.32, no.8, pp.38-46, 1995.

[5] Chen, T-F., 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] Ding, C., Kennedy, K.: "The memory bandwidth bottleneck and its amelioration by a compiler"; Proc. 14-th Int. Parallel and Distributed Processing Symp., Cancun, Mexico, pp.181-189, 2000.

[7] Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L., Tullsen, D.M., "Simultaneous multithreading – a platform for next generation processors"; IEEE Micro, vol.17, no.5, pp.12-19, 1997.

[8] Govindarajan, R., Suciu, F., 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.

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

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

[11] Jensen, K., "Coloured Petri nets"; in: "Advanced Course on Petri Nets 1986" (Lecture Notes in Computer Science 254), Rozenberg, G. (ed.), pp.248-299, Springer–Verlag 1987.

[12] Klaiber, A.C., 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.

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

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

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

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

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

[18] Tullsen, D.M., Eggers, S.J., Levy, H.M., "Simultaneous multithreading: maximizing on-chip parallelism"; Proc. 22-nd Annual Int. Symp. on Computer Architecture (ISCA'22), Santa Margherita Ligure, Italy, pp.392-403, 1995.

[19] 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.

[20] 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.

[21] Zuberek, W.M., "Approximate performance evaluation of multithreaded distributed memory architectures"; Proc. 15-th Performance Engineering Workshop, Bristol, UK, pp.81-92, 1999.