**Hardware Design and Petri Nets**, A. Yakovlev et.al. (eds.), ch.16, pp.311–331.
 (DOI 10.1007/978-1-4757-3143-9_16).

Chapter 16

# PERFORMANCE MODELING OF MULTITHREADED DISTRIBUTED MEMORY ARCHITECTURES

Wlodek M. Zuberek
*Department of Computer Science*
*Memorial University of Newfoundland*
*St.John's, NF, Canada A1B 3X5*
wlodek@cs.mun.ca

**Abstract** In multithreaded distributed memory architectures, long–latency memory operations and synchronization delays are tolerated by suspending the execution of the current thread and switching to another thread, which is executed concurrently with the long–latency operation of the suspended thread. Timed Petri nets are used to model several multithreaded architectures at the instruction and thread levels. Model evaluation results are presented to illustrate the influence of different model parameters on the performance of the system.

**Keywords:** multithreaded architectures, distributed memory architectures, performance modeling, timed Petri nets

## 1. INTRODUCTION

The performance of microprocessors has been steadily improving over the last two decades, doubling every 18 months. On the other hand, the memory performance and the performance of the processor interconnecting networks have not improved nearly as fast. The mismatch of performance between the processor and the memory subsystem significantly impacts the overall performance of both uniprocessor and multiprocessor systems. Recent studies have shown that the number of processor cycles required to access main memory doubles approximately every six years [**?**]. This growing gap between the processing power of modern micro-

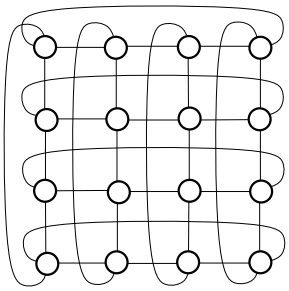

Figure 16.1. Outline of a 16–processor system.

processors and the memory latency significantly limits the performance of each node in multicomputer systems. Trends in processor technology and memory technology indicate that this gap will continue to widen at least for the next several years. It is not unusual to find that the processor is stalled 60% of time waiting for the completion of memory operations [**?**].

In distributed memory systems, the latency of memory accesses is much more pronounced as memory access requests may need to be forwarded through several intermediate nodes before they reach their destination, 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.

Instruction reordering is one of the approaches used to alleviate the problem of divergent processor and memory performances. Multithreading is another approach which combines software (compilers) and hardware (multiple thread contexts) means [**?**, **?**, **?**].

Multithreading is an architectural approach to tolerating long–latency memory accesses and synchronization delays in distributed memory systems. The general idea is quite straightforward. When a long–latency memory operation occurs, the processor instead of waiting for its completion (which in distributed memory systems can easily require a hundred or more processor cycles), switches to another thread if such a thread is ready for execution. Switching to another thread can be performed very efficiently because the threads are executing in the same address space. If different threads are associated with different sets of processor registers, switching from one thread to another can be done in one or just a few processor cycles [**?**, **?**].

A distributed memory system with 16 processors connected by a 2–dimensional torus–like network is used as a running example in this paper; an outline of such a system is shown in Fig.16.1.

It is usually assumed that the messages 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

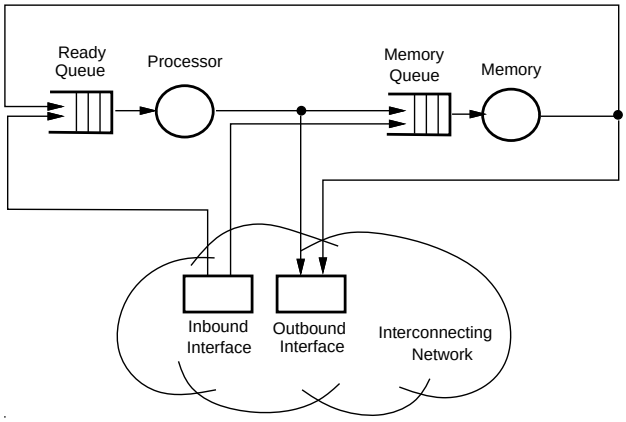


Figure 16.2. Outline of a single multithreaded processor.

between two nodes, each of them is equally likely to be used. The average length of the shortest path between two nodes, or the average number of hops (from one node to another) that a message must perform to reach its destination, is usually determined assuming that the memory accesses are uniformly distributed over the nodes of the system.

Although many specific details refer to this 16–processor system, most of them can easily be adjusted to other systems by changing the values of a few parameters. Some of these adjustments are discussed in the concluding remarks.

Each node in the network shown in Fig.16.1 is a multithreaded processor which contains a processor, local memory, and two network interfaces, as shown in Fig.16.2. 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.16.2 also shows a queue of ready threads; whenever the processor performs a context switch (i.e., switch from one thread to another), a thread from this queue is selected for execution and the execution continues until another context switch is performed.

Switching from one thread to another can take place:

**(a)** for each long–latency memory access [**?**] (a typical approach when context switching can be done very quickly),

**(b)** for each long–latency remote memory access [**?**] (a typical approach when the time of context switching is comparable with the memory cycle; in this case the processor is stalled while the thread accessing local memory waits for the result),

**(c)** after every instruction [**?**] (this approach is advantageous for eliminating data dependencies which slow–down the pipeline; since consecutive instructions are from different threads, they have no data dependencies); typically the number of threads is equal to the number of pipeline stages, so no inter-instruction dependencies can stall the pipeline.

For the first two cases, when a context switch is performed as a result of long–latency memory access, the current thread becomes 'suspended', its request is directed to the memory module (through the interconnecting network), and when the result of this request is received, the thread becomes 'ready' again and joins the queue of ready threads waiting for another execution phase on the processor.

For case (c), the thread, after issuing a long–latency memory access request, becomes 'waiting' for the result of the requested operation. If a waiting thread is selected for execution, its 'slot' simply remains empty (i.e., no instruction is issued), which is equivalent to a single–cycle pipeline stall. Since the threads issue their instructions one ofter another, only a few processor cycles are lost during a long–latency operation of a single thread.

The average number of instructions executed between long–latency operations (and context switches) is called the runlength of a thread, $\ell_t$, and is one of important 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, $p_\ell$, (or remote, $p_r = 1 - p_\ell$) memory; as the value of $p_\ell$ decreases (or $p_r$ increases), the effects of communication overhead and congestion in the interconnecting network (and its switches) become more pronounced; for $p_\ell$ 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 this 'limiting' component (called the bottleneck) also allows to estimate the performance of the system.

## 2. MODELS

Petri nets have become a popular formalism for modeling systems that exhibit parallel and concurrent activities [**?**, **?**]. In timed nets [**?**],

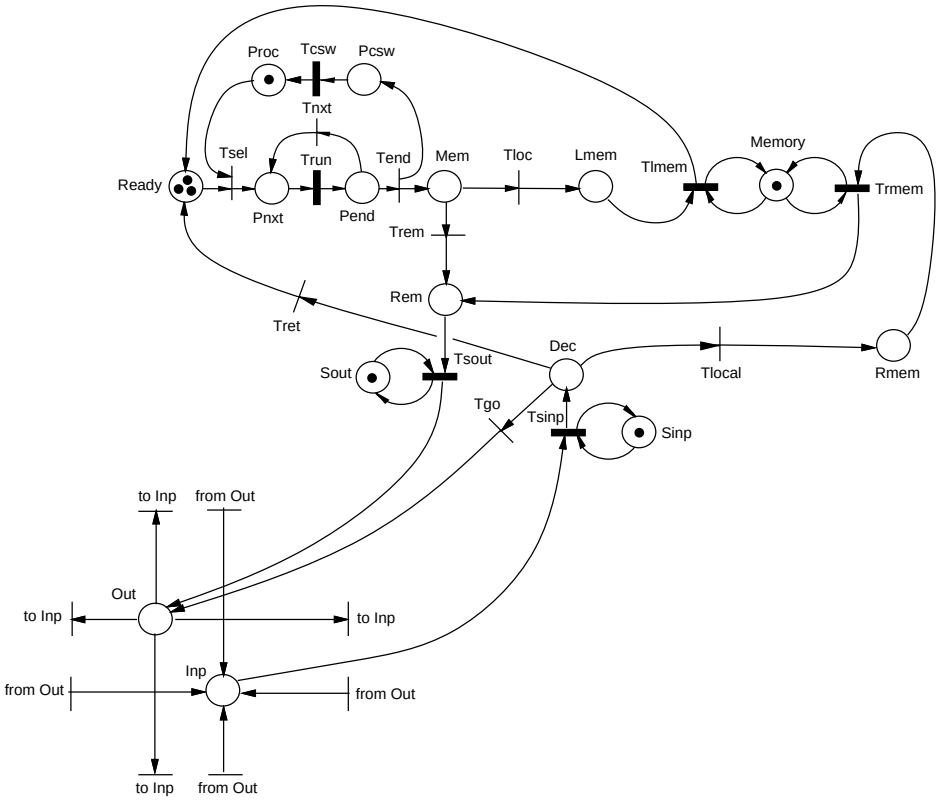


Figure 16.3. Instruction–level Petri net model of a multithreaded processor; case (a).

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.16.3.

The execution of each instruction of the 'running' thread is modeled by transition $Trun$. 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, $n_t$. It is assumed that this number does not change in time.

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 reflecting the runlength, $\ell_t$, of the thread. In general, the free–choice probability assigned to $Tnxt$ is equal to $(\ell_t - 1)/\ell_t$, so if $\ell_t$ is equal to 10, the probability of $Tnxt$ is 0.9; if $\ell_t$ is equal to 5, this probability is 0.8, and so on. The free–choice probability of $Tend$ is just $1/\ell_t$.

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 $Pcsw$. Firing the timed transition $Tcsw$ represents the context switching. When it 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, 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 accesses 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 $Tloc$, $p_\ell$, is the probability of long–latency accesses to local memory; the free–choice probability of $Trem$ is $p_r = 1 - p_\ell$.

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 $Tlocal$, which represents remote requests accessing the memory at the destination node. In the last case, the 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 $Tlocal$ characterize the interconnecting network [**?**].

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 $Trem$).

Instruction dependencies, within each thread, occasionally stall the pipeline to delay the execution of an instruction until its argument is available. Some dependencies can be removed by reordering the instructions or by renaming the registers (either by compiler or by hardware in

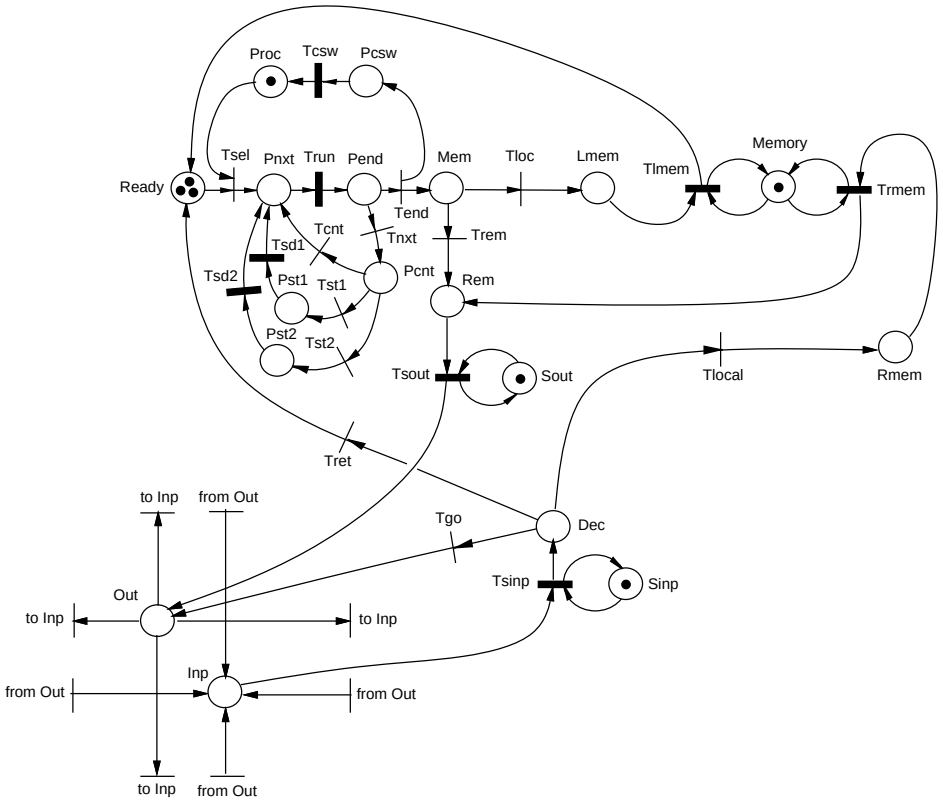


Figure 16.4. Instruction–level Petri net model of a multithreaded processor; case (a) with pipeline delays.

the instruction issue unit); the remaining dependencies are detected in the pipeline, and they stall the pipeline for one or more processor cycles. Pipeline stalls are not represented in Fig.16.3.

Fig.16.4 shows a modification of the model from Fig.16.3 in which the transition $Tnxt$ is augmented by a free–choice place $Pcnt$ with three choices: $Tcnt$ which represents continuation without stalling, $Tst1$ which introduces a single–cycle stall (timed transition $Tsd1$), and $Tst2$ which introduces a two–cycle stall (timed transition $Tsd2$).

The choice probabilities associated with these three transitions characterize the frequency and the durations of pipeline stalls. In evaluation of this model, these choices are described by two probabilities, $p_{s1}$ and $p_{s2}$, associated with $Tst1$ and $Tst2$, respectively (the probability associated with $Tcnt$ is simply $1 - p_{s1} - p_{s2}$). Although only two cases of pipeline stalls are represented in Fig.16.4, other cases can be modeled in a similar way.

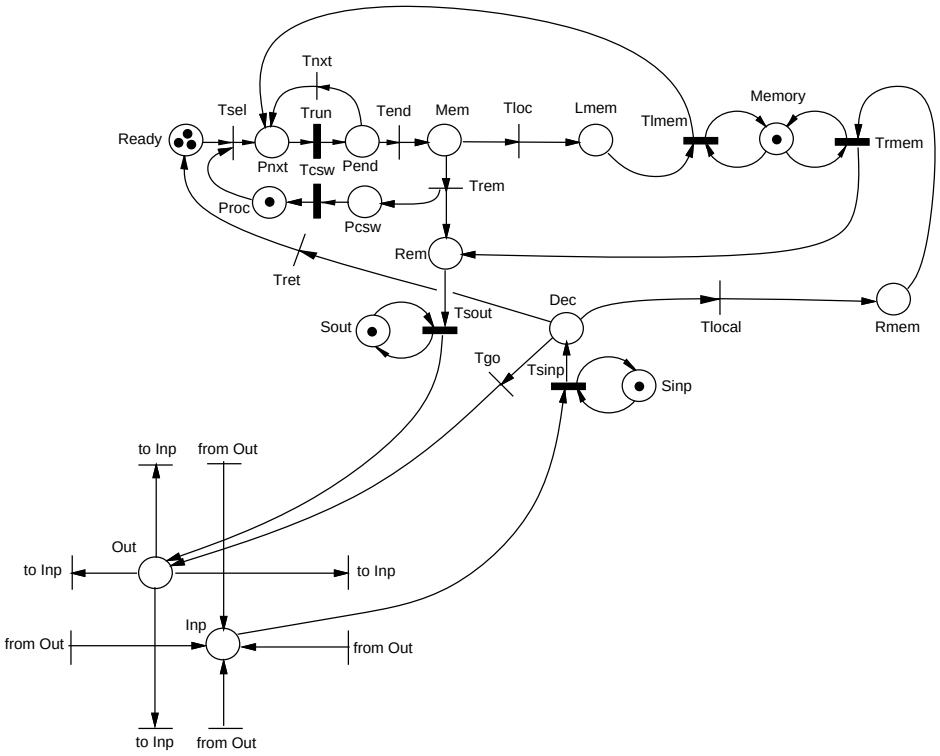


Figure 16.5. Instruction–level Petri net model of a multithreaded processor; case (b).

Fig.16.5 shows a processor's model for the case when the context switching is performed for remote memory accesses only (case (b)). In this case, if the long–latency request is to local memory ($Tloc$), the processor 'waits' for the completion of the memory access and then continues the execution of the same thread. If the request is to remote memory ($Trem$), the processor performs context switch and executes another thread(s) concurrently with the remote memory access.

Fig.16.5 does not represent the pipeline stalls; if needed, they can be added in the same way as in Fig.16.4.

Petri net model of fine–grain multithreading (case (c)), in which consecutive instructions are (cyclically) issued from consecutive threads, is more elaborate because of the representation of the cyclic thread selection. An outline of the model is shown in Fig.16.6, with a shaded area that models the switching from one thread to another (and suspending the threads during their long–latency operations). It should be observed that this outline is basically the same as in Fig.16.3, Fig.16.4 and Fig.16.5.

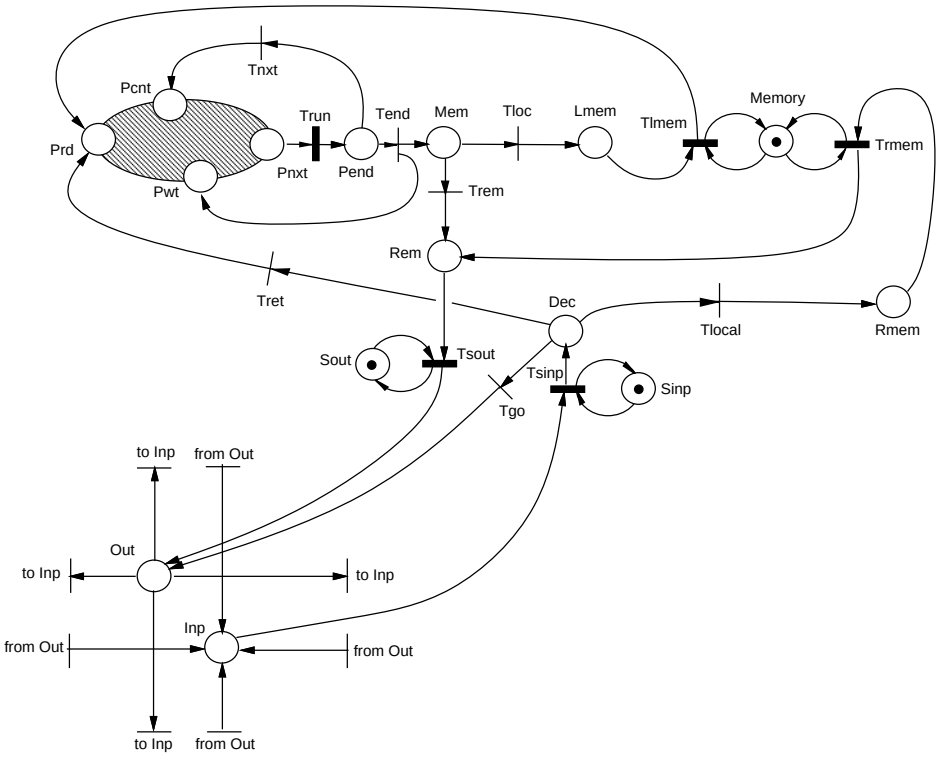


Figure 16.6. Instruction–level Petri net model of a multithreaded processor; case (c).

The detailed representation of the shaded area in Fig.16.6 (with the adjacent elements), for a processor with four threads, is shown in Fig.16.7. Fig.16.7 may seem to be a bit complicated, but it has a regular structure that is repeated for each of the four threads (and would also be used for additional threads).

This basic structure, for thread "2", is shown in Fig.16.8. The idea of this model is as follows. If the thread is "ready", a token is waiting in $Pth2$ for a "control token" to appear in $Ps2$ (the marking of $Ps2$ in Fig.16.8 indicates that an instruction from thread "2" is going to be issued in the next processor cycle). Place $Ps2$ is an element of a "thread ring" (in Fig.16.7 this ring connects $Ps1$, $Ps2$, $Ps3$, $Ps4$ and back to $Ps1$, and there are several different ways connecting consecutive threads). This "thread ring" contains a single token ($Ps1$ in Fig.16.7 and $Ps2$ in Fig.16.8).

If the selected thread is "ready", the firing of $Tth2$ inserts a token in $Pnxt$ (the next instruction to be executed by $Trun$), and also a token in $Pr2$. If the issued instruction does not perform long–latency memory access, the free–choice transition $Tnxt$ is fired (with the probability de-

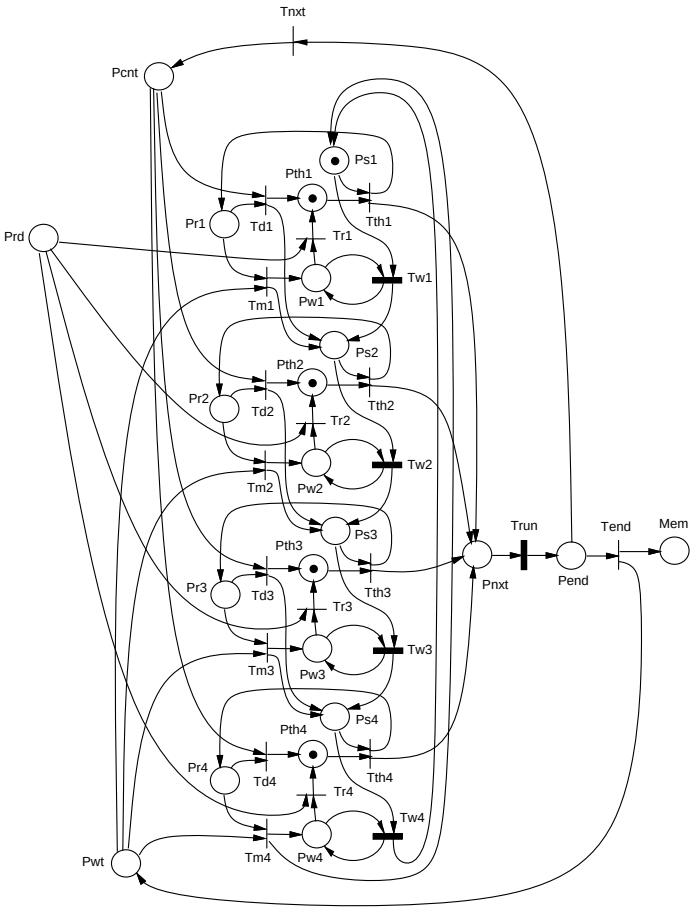


Figure 16.7. Thread switching in a fine-grain multithreaded processor.

pending upon the thread runlength $\ell_t$), and a token is deposited in $Pcnt$. This token (together with a token in $Pr2$) enables firing of $Td2$, which regenerates a token in $Pth2$ and forwards the control token to $Ps3$.

If transition $Tend$ is selected for firing rather than $Tnxt$, a long–latency memory access (local or remote) is initiated, and a token is deposited in $Pwt$. In this case $Tm2$ is enabled to fire, which inserts a token in $Pw2$ (to indicate that the thread is waiting for termination of its long-latency memory access), and also the control token is forwarded to $Ps3$.

If a thread is "waiting" and a selection token appears in $Ps2$, the timed transition $Tw2$ fires and, after a unit of time (one processor cycle), deposits a control token in $Ps3$.

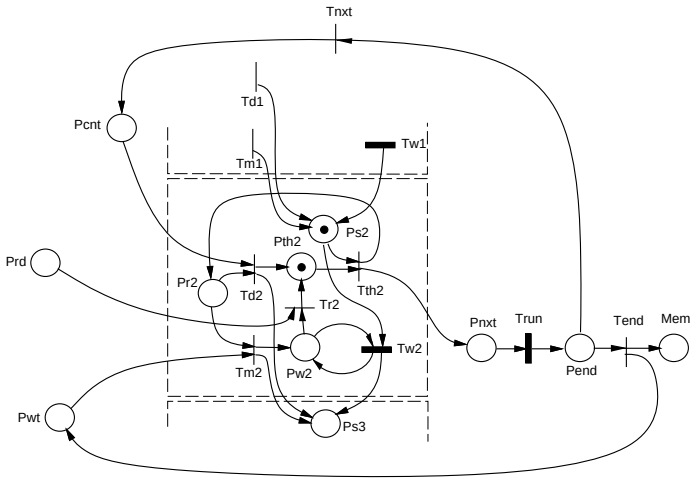


Figure 16.8. Single thread switching section.

Finally, when the long–latency memory operation (local or remote) is completed, a token is deposited in $Prd$ (Fig.16.6), and the "waiting" thread becomes "ready" by firing $Tr2$.

## 3. PERFORMANCE

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 values used in subsequent evaluations are as follows:

| *symbol* | *parameter* | *values* |
|---|---|---|
| $n_t$ | the (average) number of threads | 2,...,20 |
| $\ell_t$ | thread runlength | 5,10,20 |
| $t_{cs}$ | context switching time | 1,2,5 |
| $t_m$ | memory cycle time | 10 |
| $t_s$ | switch delay | 5,10 |
| $p_\ell$, $p_r$ | probability of accesses to local/remote memory | 0.1,...,0.9 |
| $p_{s1}$, $p_{s2}$ | probabilities of pipeline stalls | 0.1,0.2 |

Fig.16.9 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, $p_\ell$, for fixed values of other modeling parameters. It can be observed that, for values of $p_\ell$ close to 1, the utilization increases with the number of available threads $n_t$, and tends to the bound

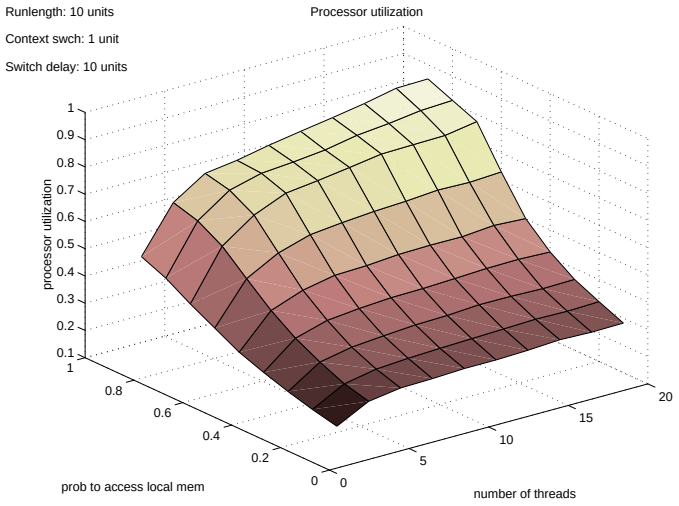


Figure 16.9. Processor utilization; $t_{cs} = 1$, $\ell_t = 10$, $t_s = 10$.

0.91 which is determined, in this case, by the ratio of $\ell_t/(\ell_t + t_{cs})$ (the context switching time, $t_{cs}$, is an overhead of multithreading).

For smaller values of $p_\ell$, 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.

The bottlenecks can be identified by comparing service demands for the different components and the system [**?**]; the component with the highest service demand is the first to reach its 'saturation' (i.e., utilization of almost 100%) which limits the performance of all other elements of the system.

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

| *component* | *service demand* |
|---|---|
| processor | $\ell_t$ |
| memory | $t_m$ |
| inbound switch | $2 * p_r * n_h * t_s$ |
| outbound switch | $2 * p_r * t_s$ |

where $n_h$ is the average number of hops (in the interconnecting network) that a request must perform to reach its destination (for a 16–processor system, with a uniform distribution of accesses over the nodes, the value of $n_h$ is close to 2 [**?**]; 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 $p/2$).

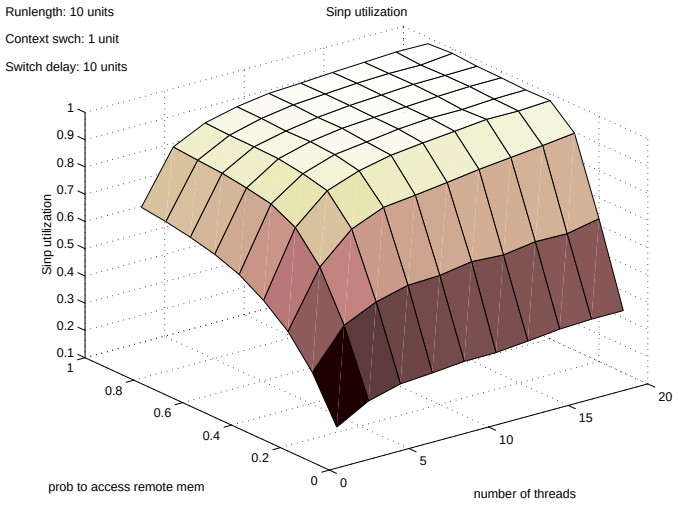


Figure 16.10. Inbound switch utilization; $t_{cs} = 1$, $\ell_t = 10$, $t_s = 10$.

If $\ell_t = t_m = t_s$, and $n_h = 2$, the inbound switch becomes the bottleneck for $p_r > 0.25$; for $p_r < 0.25$ the processor and the memory are the bottlenecks (their service demands are equal for $t_m = \ell_t$).

Fig.16.10 shows the utilization of the inbound switch (for the same values of modeling parameters as in Fig.16.9); it should be noted that in Fig.16.10 (as well as in Fig.16.11) the probability of accessing remote memory, $p_r$, is used instead of $p_\ell$, so the 'front part' of Fig.16.10 corresponds to the 'back part' of Fig.16.9 and vice versa.

Fig.16.10 shows that the inbound switch enters its saturation quite quickly for increasing values of $n_t$ and $p_\ell$. The delay introduced by the inbound switch is simply too large if the probability of accesses to remote memory, $p_r$, can be greater than 0.25.

Fig.16.11 shows the utilization of the inbound switch for the switch delay $t_s = 5$. In this case the 'saturated' region is much smaller than in Fig.16.10. The corresponding utilization of the processor is shown in Fig.16.12; the utilization is significantly better than in Fig.16.9, but the limiting effects of the inbound switch can still be observed for small values of $p_\ell$.

The influence of pipeline stalls is rather straightforward to predict; since the stalled processor cycles are lost, the performance of the processor must decrease when pipeline stalls are taken into account.

Fig.16.13 and Fig.16.14 show the utilization of the processor (as a function of the number of threads, $n_t$, and the probability of accesses to local memory, $p_\ell$, as before), for two different probabilities of pipeline stalls. Fig.16.13 corresponds to the case when $p_{s1} = p_{s2} = 0.1$, i.e., for 10% of instructions the pipeline stalls for one cycle, and for another

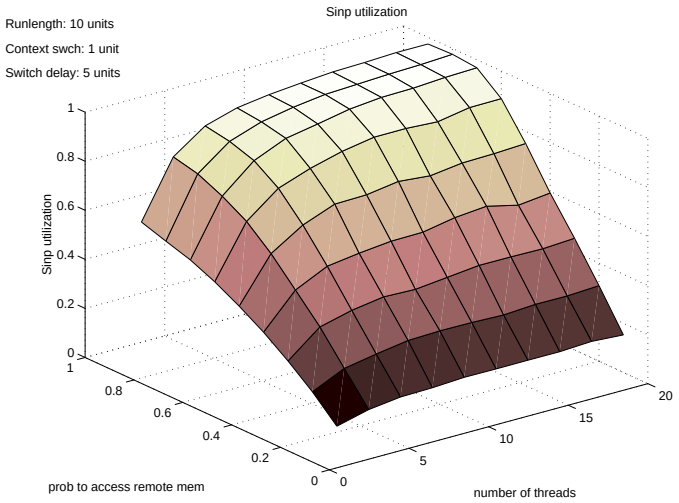


Figure 16.11. Inbound switch utilization; $t_{cs} = 1$, $\ell_t = 10$, $t_s = 5$.

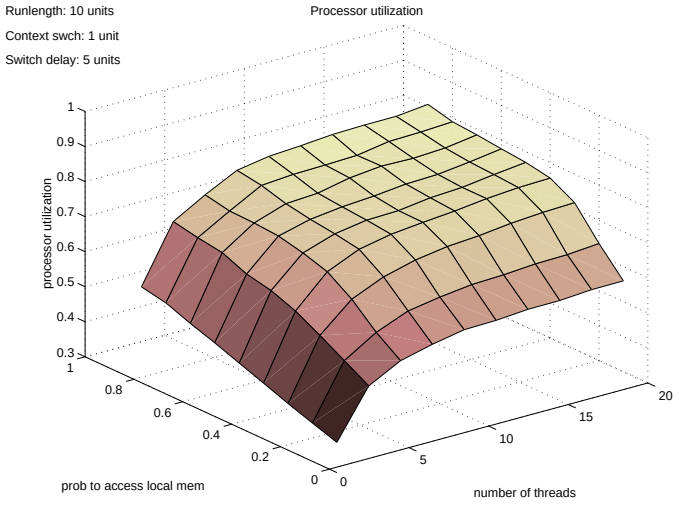


Figure 16.12. Processor utilization; $t_{cs} = 1$, $\ell_t = 10$, $t_s = 5$.

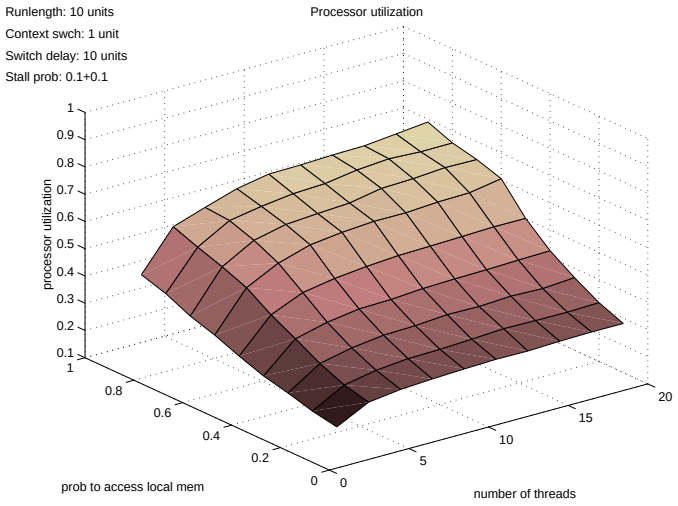


Figure 16.13. Processor utilization; $t_{cs} = 1$, $\ell_t = 10$, $t_s = 10$, $p_{s1} = p_{s2} = 0.1$.

10% of instructions the pipeline is stalled for two processor cycles. Consequently, the processor is stalled approximately 25% of time, so its utilization is expected to be approximately 25% lower than in the case without pipeline delays (Fig.16.9). Indeed, a comparison of Fig.16.13 and Fig.16.9 shows such a difference for the values of $p_\ell$ close to 1. For $p_\ell < 0.75$ (or $p_r > 0.25$), the inbound switch is the bottleneck, so there is no significant difference between Fig.16.13 and Fig.16.9 because in both cases the performance is determined by the delay of the inbound switch (which is the same for both cases).

Fig.16.14 shows the utilization of the processor which is stalled approximately 40% of time ($p_{s1} = p_{s2} = 0.2$). It should be noted that, for the values of $p_\ell$ close to 1, the utilization is further reduced, as expected, while the utilization in the "saturated region" (i.e., for small values of $p_\ell$) is not affected by the pipeline stalls, and remains the same as in Fig.16.13 and Fig.16.9.

The influence of the context switching time on the processor utilization is shown in Fig.16.15 and Fig.16.14. Because the time of context switching is an overhead for the thread execution, the upper bound on the processor utilization is $\ell_t/(\ell_t + t_{cs})$. For $\ell_t = 10$ and $t_{cs} = 1$ (Fig.16.9), this bound is 0.91. For $\ell_t = 10$ and $t_{cs} = 2$ (Fig.16.15), this bound becomes 0.83, and for $\ell_t = 10$ and $t_{cs} = 5$ (Fig.16.16), the upper bound decreases to 0.67. The influence of these upper bounds can easily be observed in Fig.16.15 and Fig.16.16 for the values of $p_\ell$ close to 1.

Similarly to the probability of pipeline stalls, the context switching time has little effect on processor utilization when the performance is limited by the inbound switch (i.e., for small values of $p_\ell$).

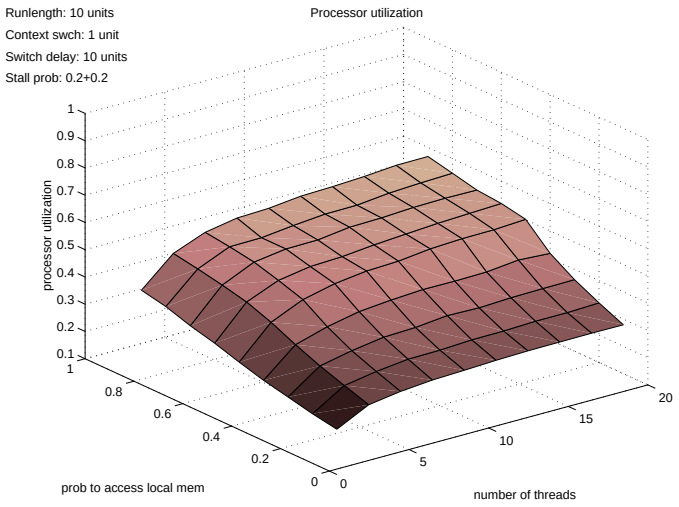


Figure 16.14. Processor utilization; $t_{cs} = 1$, $\ell_t = 10$, $t_s = 10$, $p_{s1} = p_{s2} = 0.2$.

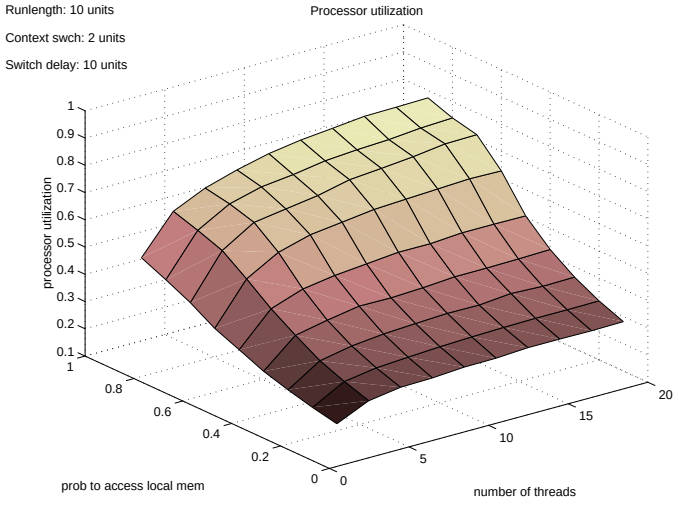


Figure 16.15. Processor utilization; $t_{cs} = 2$, $\ell_t = 10$, $t_s = 10$.

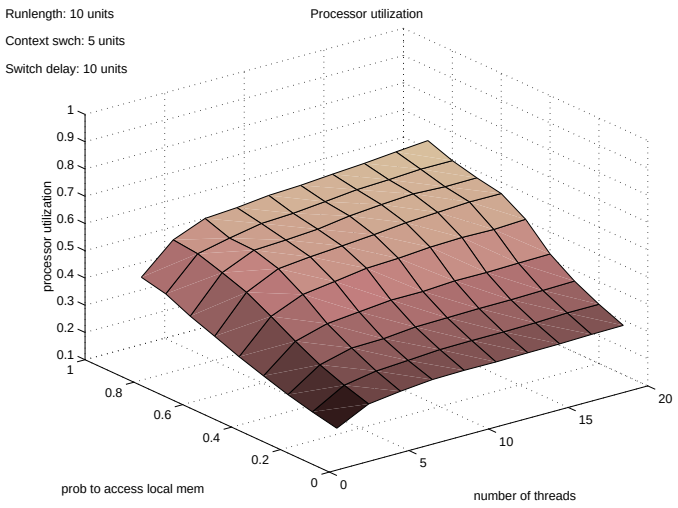


Figure 16.16. Processor utilization; $t_{cs} = 5$, $\ell_t = 10$, $t_s = 10$.

The combined effect of pipeline stalls and the context switching time is not difficult to predict. Since both factors reduce the number of processor cycles used for execution of program instructions, their effects are cumulative with respect to reducing the utilization of the processor.

If context switching is performed for long–latency remote memory accesses (case (b)), the context switching time $t_{cs}$ (comparable to $t_m$ in this case) introduces a considerable overhead, so the upper bound on the utilization of the processor, for $t_{cs} = t_m$, is $\ell_t/(\ell_t + t_{cs})$; for $t_{cs} = t_m = \ell_t$, this upper bound is equal to 0.5; for $\ell_t = 20$ and $t_{cs} = t_m = 10$, the bound increases to 0.67. Consequently, much lower values of processor utilization are obtained in this case; to achieve a reasonable performance, large values of the runlength, $\ell_t$, are needed.

## 4. CONCLUDING REMARKS

All Petri net models of multithreaded distributed memory architectures discussed in this paper have a common part that represents the memory and the interconnecting network; the differences between models are in the context switching part and in the nature of context switching. Fig.16.6 shows the general 'framework' of models with the shaded area either representing case (c), as in Fig.16.7, or case (a) which could easily be extracted form Fig.16.3 or Fig.16.4; a few more model elements should be included in the shaded area to also cover case (b).

For performance analysis of derived models, the interconnecting network is characterized by the average number of hops, $n_h$. Consequently, different networks characterized by the same value of $n_h$ will yield the

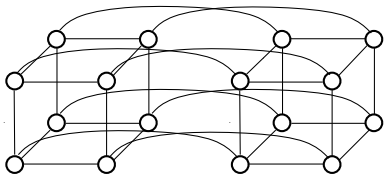

Figure 16.17. Outline of a 16–processor system.

same performance characteristics of the nodes. For example, Fig.16.17 shows a hypercube network for a 16–processor system that can be used instead of a 2–dimension torus network shown in Fig.16.1. Since each node in Fig.16.17 is connected to 4 neighbors (as is the case in Fig.16.1), the average numbers of hops (with the same assumptions as before) for the two networks are the same, and then the performance characteristics for the two types of interconnecting networks are identical.

Moreover, models of systems with different numbers of processors (e.g., 25, 36, etc.) require only minor adjustment of a few model parameters (the free–choice probabilities describing the traffic of messages in the interconnecting network); otherwise the models are as presented in this paper.

One of the assumptions of the derived models was 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 made 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 (especially for values of $p_r$ close to 1).

In many cases the presented models can be abstracted to a less detailed form without any significant loss of accuracy. One of such transformations replaces the instruction–level model of a processor by its thread–level model; Fig.16.18 shows such a transformation applied to the model shown in Fig.16.3. In Fig.16.18 the firing time of $Trun$ is exponentially distributed with the average value equal to the thread runlength; it represents the (random) execution time of a thread (with no individual instructions). It appears that this simplification does not affect the results in any significant way, but it speeds up (several times) the simulation of the model, eliminating many events which do not actually contribute to the performance characteristics.

Petri net models of multiprocessor systems contain many 'regularities' which can be used for model reduction in a more sophisticated modeling formalism. For example, in colored Petri nets [**?**], tokens are associated with attributes (called colors), so different activities can be associated

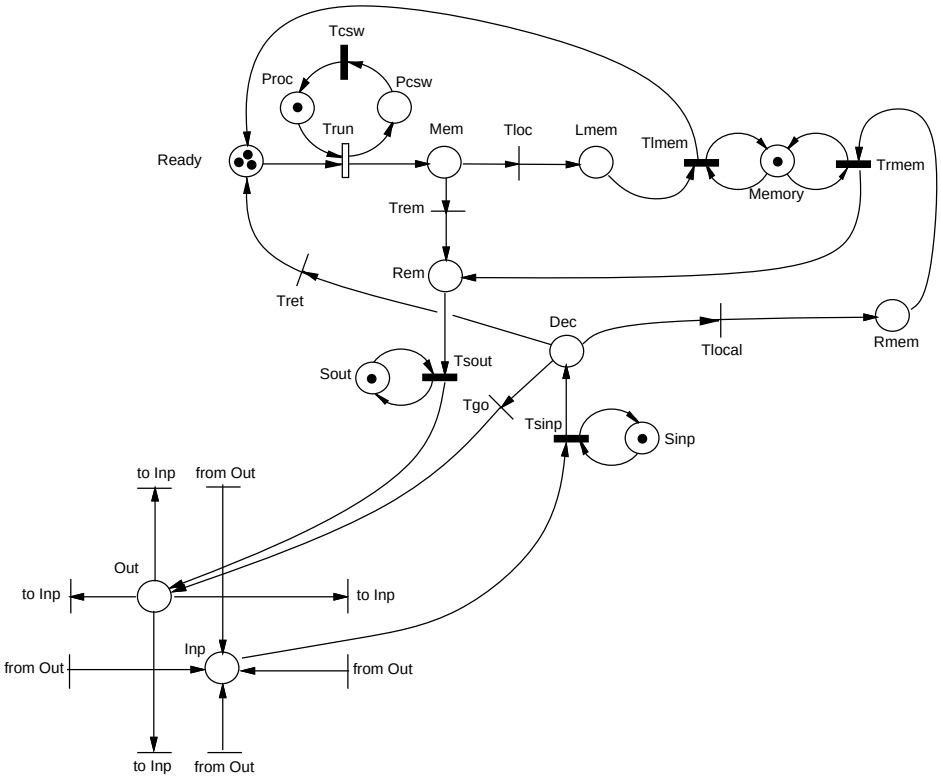


Fig.16.18. Thread–level Petri net model of a multithreaded processor, case (a).

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). Similarly, the thread sections in a model of multithreading shown in Fig.17.7 can also be represented by colors, additionally simplifying the model. Some other aspects of colored net models are discussed in [**?**].

## Acknowledgments

The Natural Sciences and Engineering Research Council of Canada partially supported this research through Research Grant OGP8222.

Collaboration with Dr. R. Govindarajan of the Indian Institute of Science in Bangalore, India, is gratefully acknowledged.

Several remarks of two anonymous referees were helpful in revising and improving the paper.

## 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] Agrawal, A., Lim, B-H., Kranz, D., Kubiatowicz, J., "April: a processor architecture for multiprocessing"; Proc. 17-th Annual Int. Symp. on Computer Architecture, pp.104-114, 1990.

[3] Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Posterfield, A., Smith, B., "The Tera computer system"; Proc. Int. Conf. on Supercomputing, Amsterdam, The Netherlands, pp.1-6, 1990.

[4] Boland, K., Dolles, A., "Predicting and precluding problems with memory latency"; IEEE Micro, vol.14, pp.59-67, 1994.

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

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

[7] Govindarajan, R., Nemawarkar, S.S., LeNir, P., "Design and performance evaluation of a multithreaded architecture"; Proc. First IEEE Symp. on High–Performance Computer Architecture, Raleigh, NC, pp.298-307, 1995.

[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 (PNPM'97), St. Malo, France, pp.153-162, 1997.

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

[10] 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.

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

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

[13] Sinharoy, B., "Optimized thread creation for processor multithreading"; Computer Journal, vol.40, no.6, pp.388-399, 1997.

[14] Smith, B.J., "Architecture and applications of the HEP multiprocessor computer System"; Proc. SPIE – Real-Time Signal Processing IV, vol. 298, pp. 241–248, 1981.

[15] Weber, W.D., Gupta, A., "Exploring the benefits of multiple contexts in a multiprocessor architecture: preliminary results"; Proc. 16-th Annual Int. Symp. on Computer Architecture, pp.273-280, 1989.

[16] 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.

[17] Zuberek, W.M., Govindarajan, R., "Performance balancing in multithreaded multiprocessor architectures"; Proc. 4-th Australasian Conf. on Parallel and Real–Time Systems (PART'97), Newcastle, Australia, pp.15-26, 1997.