Chapter 13

# Timed Petri Nets in Performance Exploration of Simultaneous Multithreading

Wlodek M. Zuberek

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/48601

## 1. Introduction

In modern computer systems, the performance of the whole system is increasingly often limited by the performance of its memory subsystem [1]. Due to continuous progress in manufacturing technologies, the performance of processors has been doubling every 18 months (the so–called Moore's law [2]), but the performance of memory chips has been improving only by 10% per year [1], creating a "performance gap" in matching processor's performance with the required memory bandwidth [3]. More detailed studies have shown that the number of processor cycles required to access main memory doubles approximately every six years [4]. In effect, it is becoming more and more often the case that the performance of applications depends on the performance of the system's memory hierarchy and it is not unusual that as much as 60% of time processors spend waiting for the completion of memory operations [4].

Memory hierarchies, and in particular multi–level cache memories, have been introduced to reduce the effective latency of memory accesses [5]. 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 which tolerate long–latency memory accesses include out–of–order execution of instructions and instruction–level multithreading. The idea of out–of–order execution [1] is to execute, instead of waiting for the completion of a long–latency operation, instructions which (logically) follow the long–latency one, but which do not depend upon the result of this long–latency operation. Since out–of–order execution exploits instruction–level concurrency in the executed sequential instruction stream, it conveniently maintains code–base compatibility [6]. In effect, the instruction stream is dynamically decomposed into micro-threads, which are scheduled and synchronized at no cost in terms of executing additional instructions. Although this is desirable, speedups using out–of–order

execution on superscalar pipelines are not so impressive, and it is difficult to obtain a speedup greater than 2 using 4 or 8-way superscalar issue [7]. Moreover, in modern processors, memory latencies are so long that out–of–order processors require very large instruction windows to tolerate them.

Although ultra–wide out-of-order superscalar processors were predicted as the architecture of one-billion-transistor chips, with a single 16 or 32-wide-issue processing core and huge branch predictors to sustain good instruction level parallelism, the industry has not been moving toward the wide–issue superscalar model [8]. Design complexity and power efficiency direct the industry toward narrow–issue, high–frequency cores and multithreaded processors. According to [6]: "Clearly something is very wrong with the out–of–order approach to concurrency if this extravagant consumption of on–chip resources is only providing a practical limit on speedup of about 2."

Instruction–level multithreading [9], [10], [1] is a technique of tolerating long–latency memory accesses by switching to another thread (if it is available for execution) rather than waiting for the completion of the long–latency operation. If different threads are associated with different sets of processor registers, switching from one thread to another (called "context switching") can be done very efficiently [11], in one or just a few processor cycles.

In simultaneous multithreading [12], [6] several threads can issue instructions at the same time. If a processor contains several functional units or it contains more than one instruction execution pipeline, the instructions can be issued simultaneously; if there is only one pipeline, only one instruction can be issued in each processor cycle, but the (simultaneous) threads complement each other in the sense that whenever one thread cannot issue an instruction (because of pipeline stalls or context switching), an instruction is issued from another thread, eliminating 'empty' instruction slots and increasing the overall performance of the processor.

Simultaneous multithreading combines hardware features of wide-issue superscalar processors and multithreaded processors [12]. From superscalar processors it inherits the ability to issue multiple instructions in each cycle; from multithreaded processors it takes hardware state for several threads. The result is a processor that can issue multiple instructions from multiple threads in each processor cycle, achieving better performance for a variety of workloads.

The main objective of this work is to study the performance of simultaneously multithreaded processors in order to determine how effective simultaneous multithreading can be. In particular, an indication is sought if simultaneous multithreading can overcome the out–of–order's "barrier" of the speedup (equal to 2 [13]). A timed Petri net [14] model of multithreaded processors at the instruction execution level is developed, and performance results for this model are obtained by event–driven simulation of the developed model. Since the model is rather simple, simulation results are verified (with respect to accuracy) by state–space–based performance analysis (for those combinations of modeling parameters for which the state space remains reasonably small).

Section 2 recalls basic concepts of timed Petri nets which are used in this study. A model of simultaneous multithreading, used for performance exploration, is presented in Section 3. Section 4 discusses the results obtained by event–driven simulation of the model introduced in Section 3. Section 5 contains concluding remarks including a short comparison of simulation and analytical results.

## 2. Timed Petri nets

A marked place/transition Petri net $\mathcal{M}$ is typically defined [15] [16] as $\mathcal{M} = (\mathcal{N}, m_0)$, where the structure $\mathcal{N}$ is a bipartite directed graph, $\mathcal{N} = (P, T, A)$, with a set of places $P$, a set of transitions $T$, a set of directed arcs $A$ connecting places with transitions and transitions with places, $A \subseteq T \times P \cup P \times T$, and the initial marking function $m_0$ which assigns nonnegative numbers of tokens to places of the net, $m_0 : P \to \{0, 1, ...\}$. Marked nets can be equivalently defined as $\mathcal{M} = (P, T, A, m_0)$.

A place $p$ is an input place of a transition $t$ if the (directed) arc $(p, t)$ is in the set $A$. A place is shared if it is an input place to more than one transition. If a net does not contain shared places, the net is (structurally) conflict–free, otherwise the net contains conflicts. The simplest case of conflicts is known as a free–choice (or generalized free–choice) structure; a shared place is (generalized) free–choice if all transitions sharing it have identical sets of input places. A net is free–choice if all its shared places are free–choice. The transitions sharing a free–choice place constitute a free–choice class of transitions. For each marking function, and each free–choice class of transitions, either all transitions in this class are enabled or none of them is. It is assumed that the selection of transitions for firing within each free–choice class is a random process which can be described by "choice probabilities" assigned to (free–choice) transitions. Moreover, it is usually assumed that the random variables describing choice probabilities in different free–choice classes are independent.

All places which are not conflict–free and not free–choice, are conflict places. Transitions sharing conflict places are (directly or indirectly) potentially in conflict (i.e., they are in conflict or not depending upon a marking function; for different marking functions the sets of transitions which are in conflict can be different). All transitions which are potentially is conflict constitute a conflict class. All conflict classes are disjoint. It is assumed that conflicts are resolved by random choices of occurrences among the conflicting transitions. These random choice are independent in different conflict classes.

In timed nets [14], occurrence times are associated with transitions, and transition occurrences are real–time events, i.e., tokens are removed from input places at the beginning of the occurrence period, and they are deposited to the output places at the end of this period. All occurrences of enabled transitions are initiated in the same instants of time in which the transitions become enabled (although some enabled transitions may not initiate their occurrences). If, during the occurrence period of a transition, the transition becomes enabled again, a new, independent occurrence can be initiated, which will overlap with the other occurrence(s). There is no limit on the number of simultaneous occurrences of the same transition (sometimes this is called infinite occurrence semantics). Similarly, if a transition is enabled "several times" (i.e., it remains enabled after initiating an occurrence), it may start several independent occurrences in the same time instant.

More formally, a timed Petri net is a triple, $\mathcal{T} = (\mathcal{M}, c, f)$, where $\mathcal{M}$ is a marked net, $c$ is a choice function which assigns choice probabilities to free–choice classes of transitions or relative frequencies of occurrences to conflicting transitions (for non–conflict transitions $c$ simply assigns 1.0), $c : T \to \mathbf{R}^{0,1}$, where $\mathbf{R}^{0,1}$ is the set of real numbers in the interval [0,1], and $f$ is a timing function which assigns an (average) occurrence time to each transition of the net, $f : T \to \mathbf{R}^{+}$, where $\mathbf{R}^{+}$ is the set of nonnegative real numbers.

The occurrence times of transitions can be either deterministic or stochastic (i.e., described by some probability distribution function); in the first case, the corresponding timed nets are

referred to as D–timed nets [18], in the second, for the (negative) exponential distribution of firing times, the nets are called M–timed nets (Markovian nets [17]). In both cases, the concepts of state and state transitions have been formally defined and used in the derivation of different performance characteristics of the model [14]. Only D–timed Petri nets are used in this paper.

The firing times of some transitions may be equal to zero, which means that the firings are instantaneous; all such transitions are called *immediate* while the other are called *timed*. Since the immediate transitions have no tangible effects on the (timed) behavior of the model, it is convenient to split the set of transitions into two parts, the set of immediate and the set of timed transitions, and to fire first the (enabled) immediate transitions; only when no more immediate transitions are enabled, the firings of (enabled) timed transitions are initiated (still in the same instant of time). It should be noted that such a convention effectively introduces the priority of immediate transitions over the timed ones, so the conflicts of immediate and timed transitions should be avoided. Consequently, the free–choice and conflict classes of transitions must be "uniform", i.e., all transitions in each such class must be either immediate or timed, but not both.

Performance analysis of net models can be based on their behavior (i.e., the set of reachable states) or on the structure of the net; the former is called *reachability analysis* and the latter – *structural analysis*. For reachability analysis, the state space of the analyzed model must be finite and reasonably small while for structural analysis the model must satisfy a number of structural conditions. However, since timed Petri net models are discrete–event systems, their analysis can also be based on discrete–event simulation, which imposes very few restrictions on the class of analyzed models. All performance characteristics of simultaneous multithreading presented in Section 4 are obtained by event–driven simulation [19] of timed Petri net models shown in the next section.

## 3. Models of simultaneous multithreading

A timed Petri net model of a simple multithreaded processor is shown in Fig.1 (as usually, timed transitions are represented by solid bars, and immediate ones, by thin bars).

For simplicity, Fig.1 shows only one level of memory; this simplification is removed further in this section.

*Ready* is a pool of available threads; it is assumed that the number of of threads is constant and does not change during program execution (this assumption is motivated by steady–state considerations). If the processor is idle (place *Next* is marked), one of available threads is selected for execution (transition *Tsel*). *Cont*, if marked, indicates that an instruction is ready to be issued to the execution pipeline. Instruction execution is modeled by transition *Trun* which represents the first stage of the execution pipeline. It is assumed that once the instruction enters the pipeline, it will progress through the stages and, eventually, leave the pipeline; since these pipeline implementation details are not important for performance analysis of the processor, they are not represented here.

*Done* is another free-choice place which determines if the current instruction performs a long–latency access to memory or not. If the current instruction is a non–long–latency one, *Tnxt* occurs (with the corresponding probability), and another instruction is fetched for issuing. *Pnxt* is a free-choice place with three possible outcomes: *Tst0* (with the choice probability $p_{s0}$) represents issuing an instruction without any further delay; *Tst1* (with the

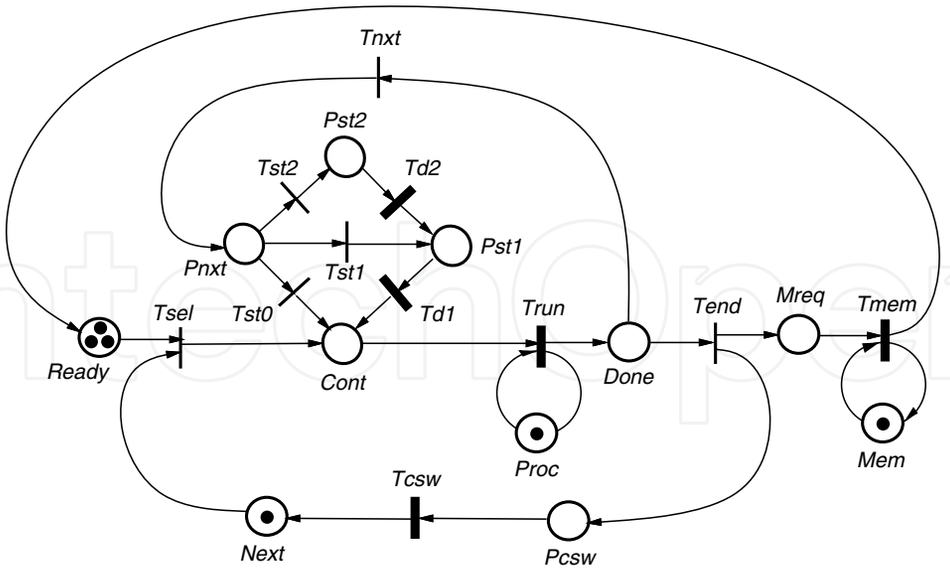


**Figure 1.** Petri net model of a multithreaded processor.

choice probability $p_{s1}$) represents a single-cycle pipeline stall (modeled by *Td1*), and *Tst2* (with the choice probability $p_{s2}$) represents a two–cycle pipeline stall (*Td2* and then *Td1*); other pipeline stalls could be represented in a similar way, if needed.

If long–latency operation is detected in the issued instruction, *Tend* initiates two concurrent actions: (i) context switching performed by enabling an occurrence of *Tcsw*, after which a new thread is selected for execution (if it is available), and (ii) a memory access request is entered into *Mreq*, the memory queue, and after accessing the memory (transition *Tmem*), the thread, suspended for the duration of memory access, becomes "ready" again and joins the pool of threads *Ready*. *Tmem* will typically represent a cache miss (with all its consequences); cache hits (at the first level cache memory) are not considered long-latency operations.

The choice probability associated with *Tend* determines the runlength of a thread, $\ell_t$, i.e., the average number of instructions between two consecutive long–latency operations; if this choice probability is equal to 0.1, the runlength is equal to 10, if it is equal to 0.2, the runlength is 5, and so on.

*Proc*, which is connected to *Trun*, controls the number of pipelines. If the processor contains just one instruction execution pipeline, the initial marking assigns a single token to *Proc* as only one instruction can be issued in each processor cycle. In order to model a processor with two (identical) pipelines, two initial tokens are needed in *Proc*, and so on.

The number of memory ports, i.e., the number of simultaneous accesses to memory, is controlled by the initial marking of *Mem*; for a single port memory, the initial marking assigns just a single token to *Mem*, for dual-port memory, two tokens are assigned to *Mem*, and so on.

In a similar way, the number of simultaneous threads (or instruction issue units) is controlled by the initial marking of *Next*.

Memory hierarchy can be incorporated into the model shown in Fig.1 by refining the representation of memory. In particular, levels of memory hierarchy can be introduced by replacing the subnet *Tmem–Mem* by a number of subnets, each subnet for one level of the hierarchy, and adding a free–choice structure which randomly selects the submodel according

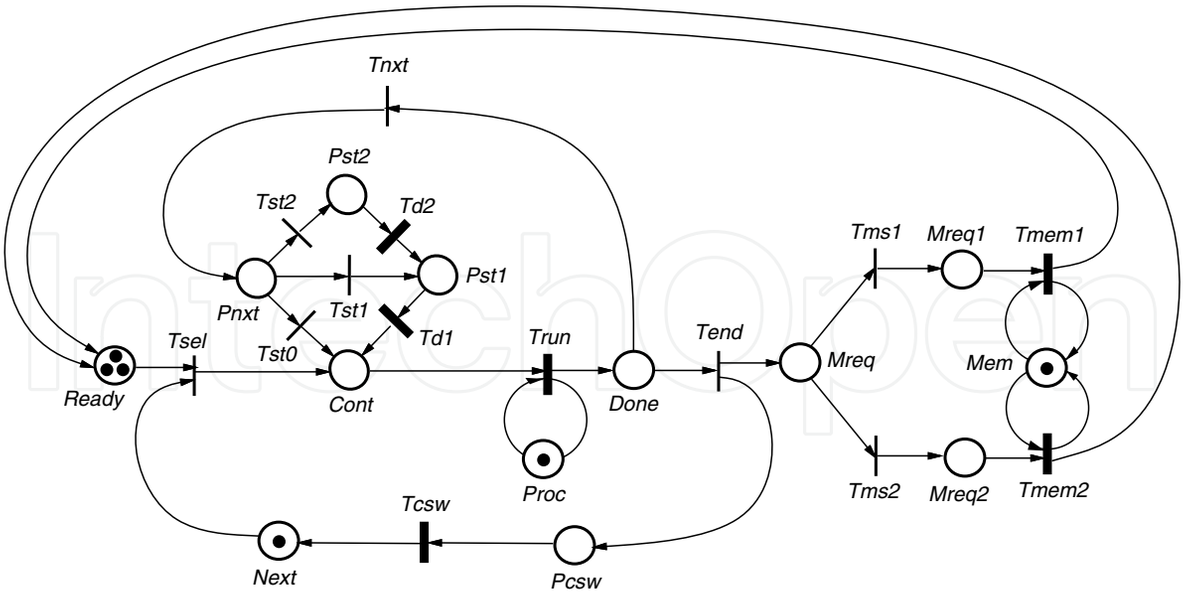


**Figure 2.** Petri net model of a multithreaded processor with a two–level memory.

to probabilities describing the use of the hierarchical memory. Such a refinement, for two levels of memory (in addition to the first-level cache), is shown in Fig.2, where *Mreq* is a free–choice place selecting either level–1 (submodel *Mem–Tmem1*) or level–2 (submodel *Mem–Tmem2*). More levels of memory can be easily added similarly, if needed.

The effects of memory hierarchy can be compared with a uniform, non–hierarchical memory by selecting the parameters in such a way that the average access time of the hierarchical model (Fig.2) is equal to the access time of the non–hierarchical model (Fig.1).

Processors with different numbers of instruction issue units and instruction execution pipelines can be described by a pair of numbers, the first number denoting the number of instruction issue units, and the second – the number of instruction execution pipelines. In this sense a 3-2 processor is a (multithreaded) processor with 3 instruction issue units and 2 instruction execution pipelines.

For convenience, all temporal properties are expressed in processor cycles, so, the occurrence times of *Trun*, *Td1* and *Td2* are all equal to 1 (processor cycle), the occurrence time of *Tcsw* is equal to the number of processor cycles needed for a context switch (which is equal to 1 for many of the following performance analyzes), and the occurrence time of *Tmem* is the average number of processor cycles needed for a long–latency access to memory.

The main modeling parameters and their typical values are shown in Table 1.

## 4. Performance exploration

The model developed in the previous section is evaluated for different combinations of modeling parameters. Performance results are obtained by event-driven simulation of timed Petri net models.

The utilization of the processor and memory, as a function of the number of available threads, for a 1-1 processor (i.e., a processor with a single instruction issue unit and a single instruction execution pipeline) is shown in Fig. 3.

| *symbol* | *parameter* | *value* |
|---|---|---|
| $n_t$ | number of available threads | 1,...,10 |
| $n_p$ | number of execution pipelines | 1,2,... |
| $n_s$ | number of simultaneous threads | 1,2,3,... |
| $\ell_t$ | thread runlength | 10 |
| $t_m$ | average memory access time | 5 |
| $t_{cs}$ | context switching time | 1,3 |
| $p_{s1}$ | prob. of one–cycle pipeline stall | 0.2 |
| $p_{s2}$ | prob. of two–cycle pipeline stall | 0.1 |

**Table 1.** Simultaneous multithreading – modeling parameters and their typical values

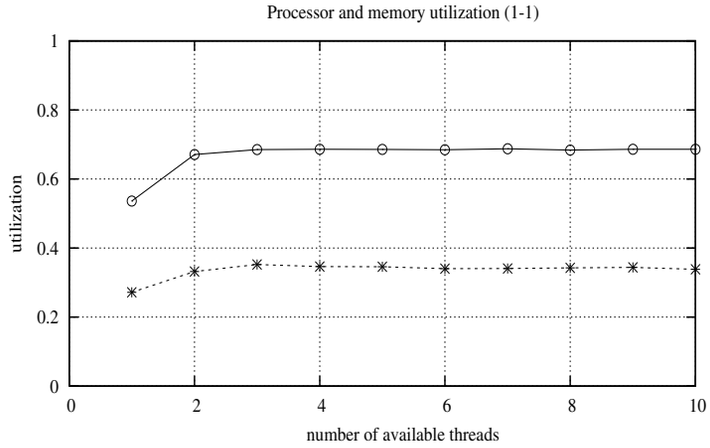


**Figure 3.** Processor (-o-) and memory (-x-) utilization for a 1-1 processor; $l_t = 10$, $t_m = 5$, $t_{cs} = 1$, $p_{s1} = 0.2$, $p_{s2} = 0.1$

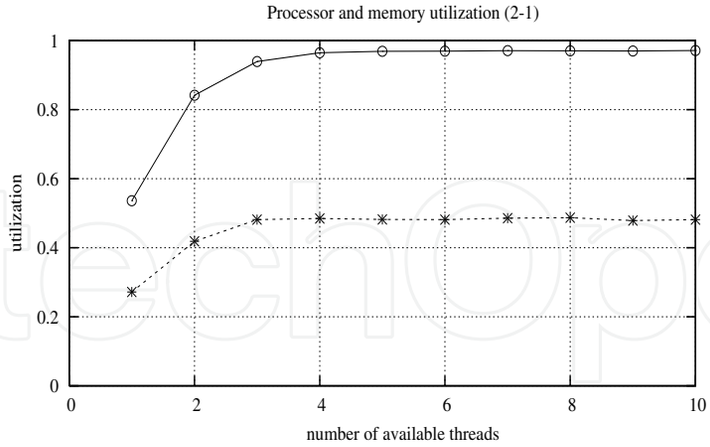


**Figure 4.** Processor (-o-) and memory (-x-) utilization for a 2-1 processor; $l_t = 10$, $t_m = 5$, $t_{cs} = 1$, $p_{s1} = 0.2$, $p_{s2} = 0.1$

The value of the processor utilization for $n_t = 1$ (i.e., for one thread) can be derived from the (average) number of unused instruction issuing slots. Since the probability of a single–cycle stall is 0.2, and probability of a two–cycle stall is 0.1, on average 40 % of issuing slots remain unused because of pipeline stalls (for all instructions except the first one in each

thread). Processor utilization for one thread is thus $\ell_t/(\ell_t + (\ell_t - 1) * 0.4 + t_m) = 10/18.6 = 0.537$, which corresponds very well with Fig.3. For a large number of threads processor utilization is obtained similarly, but with the context switching time, $t_{cs}$, replacing $t_m$, so it is $\ell_t/(\ell_t + (\ell_t - 1) * 0.4 + t_{cs}) = 0.685$.

The utilization of the processor can be improved by introducing a second (simultaneous) thread which issues its instructions in the slots unused by the first slot. Fig.4 shows the utilization of the processor and memory for a 2-1 processor, i.e., a processor with two (simultaneous) threads (or two instruction issue units) and a single pipeline. The utilization of the processor is improved by almost 50 % and is within a few percent from its upper bound (of 100 %).

The influence of pipeline stalls (probabilities $p_{s1}$ and $p_{s2}$) is shown in Fig.5 and Fig.6. Fig.5 shows that the performance actually depends upon the total number of stalls rather than specific values of $p_{s1}$ and $p_{s2}$; in Fig.5 all pipeline stalls are single–cycle ones, so $p_{s1} = 0.4$ and $p_{s2} = 0$, and the results are practically the same as in Fig. 3.

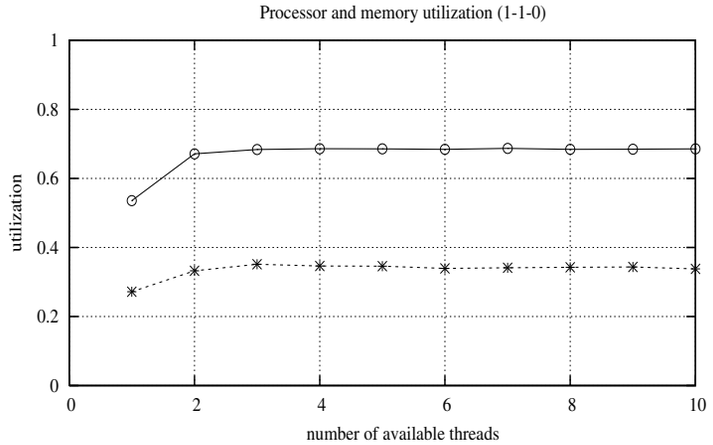


**Figure 5.** Processor (-o-) and memory (-x-) utilization for a 1-1 processor; $l_t = 10$, $t_m = 5$, $t_{cs} = 1$, $p_{s1} = 0.4$, $p_{s2} = 0$

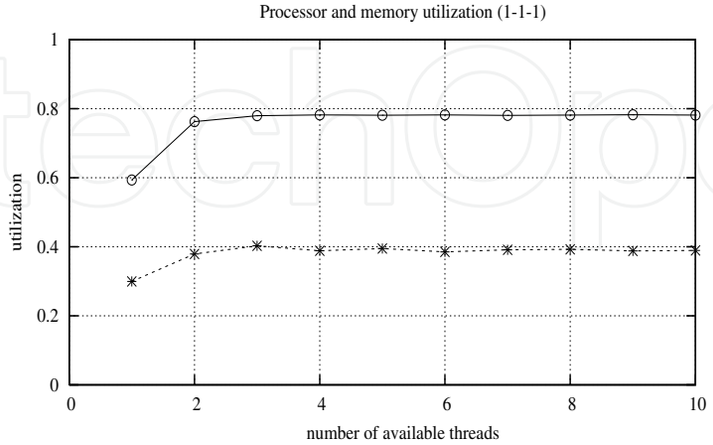


**Figure 6.** Processor (-o-) and memory (-x-) utilization for a 1-1 processor; $l_t = 10$, $t_m = 5$, $t_{cs} = 1$, $p_{s1} = 0.2$, $p_{s2} = 0$

Fig. 6 shows the utilizations of processor and memory for reduced probabilities of pipeline stalls, i.e., for $p_{s1} = 0.2$ and $p_{s2} = 0$. As is expected, the utilizations are higher than in Fig.3 and Fig.5.

A more realistic model of memory, that captures the idea of a two–level hierarchy, is shown in Fig.2. In order to compare the results of this model with Fig.3 and Fig.4, the parameters of the two–level memory are chosen in such a way that the average memory access time is equal to the memory access time in Fig.1 (where $t_m = 5$). Let the two levels of memory have access times equal to 4 and 20, respectively; then the choice probabilities are equal to 15/16 and 1/16 for level–1 and level–2, respectively, and the average access time is:

$$4 * \frac{15}{16} + 20 * \frac{1}{16} = 5.$$

The results for a 1-1 processor with a two–level memory are shown in Fig.7, and for a 2-1 processor in Fig.8.

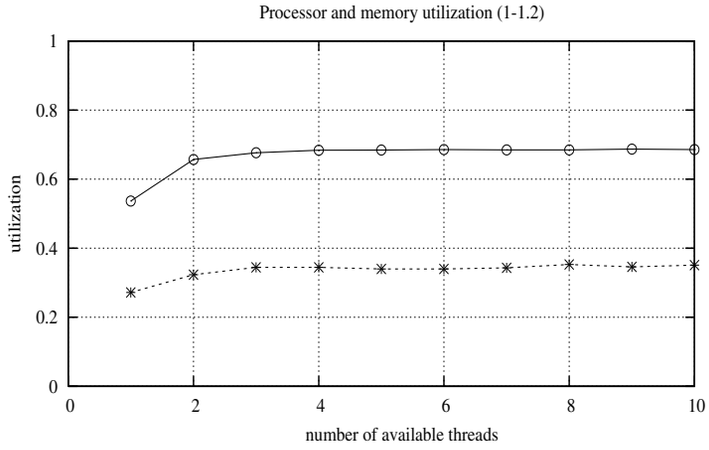


**Figure 7.** Processor (-o-) and memory (-x-) utilization for a 1-1 processor with 2-level memory; $l_t = 10$, $t_m = 4 + 20$, $t_{cs} = 1$, $p_{s1} = 0.2$, $p_{s2} = 0.1$

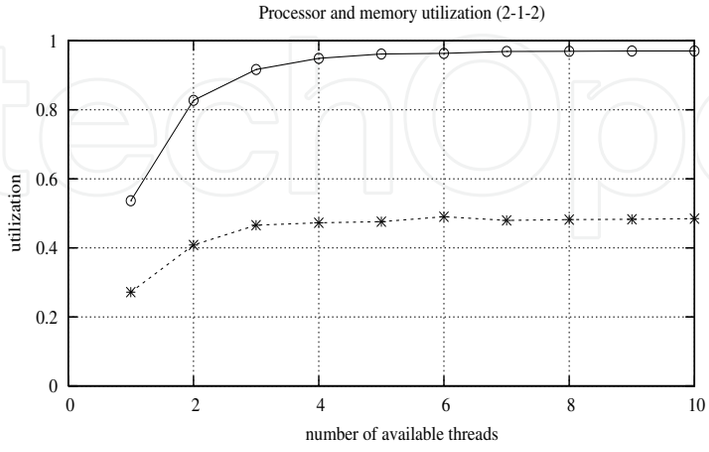


**Figure 8.** Processor (-o-) and memory (-x-) utilization for a 2-1 processor with 2-level memory; $l_t = 10$, $t_m = 4 + 20$, $t_{cs} = 1$, $p_{s1} = 0.2$, $p_{s2} = 0.1$

The results in Fig.7 and Fig.8 are practically the same as in Fig.3 and Fig.4. This is the reason that the remaining results are shown for (equivalent) one-level memory models; the multiple levels of memory hierarchy apparently have no significant effect on the performance results.

The effects of simultaneous multithreading in a more complex processor, e.g., a processor with two instruction issue units and two instruction execution pipelines, i.e., a 2-2 processor, can be obtained in a very similar way. The utilization of the processor (shown as the sum of the utilizations of both pipelines, with the values ranging from 0 to 2), is shown in Fig.9.

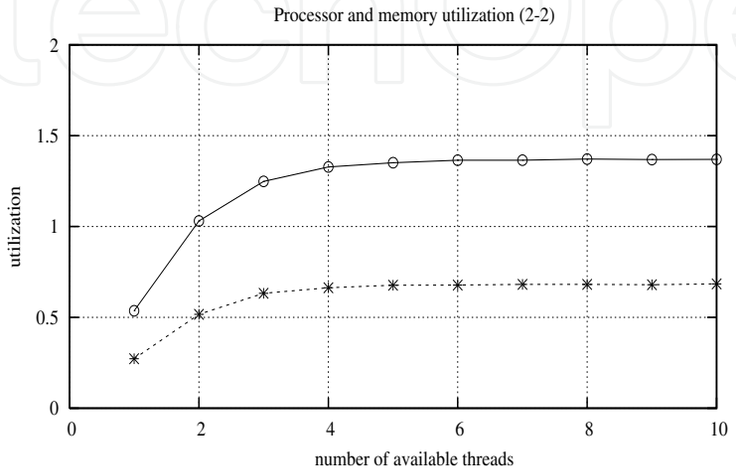


**Figure 9.** Processor (-o-) and memory (-x-) utilization for a 2-2 processor; $l_t = 10$, $t_m = 5$, $t_{cs} = 1$, $p_{s1} = 0.2$, $p_{s2} = 0.1$

When another instruction issue unit is added, the utilization increases by about 40 %, as shown in Fig.10.

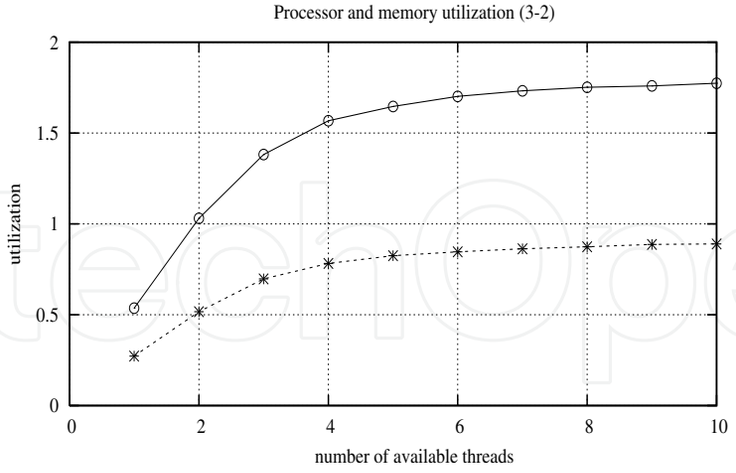


**Figure 10.** Processor (-o-) and memory (-x-) utilization for a 3-2 processor; $l_t = 10$, $t_m = 5$, $t_{cs} = 1$, $p_{s1} = 0.2$, $p_{s2} = 0.1$

Further increase of the number of the simultaneous threads (in a processor with 2 pipelines) can provide only small improvements of the performance because the utilizations of both, the processor and the memory, are quite close to their limits. The performance of the system can be improved by increasing the number of pipelines, but then the memory becomes the

system bottleneck, so its performance also needs to be improved, for example, by introducing dual ports (which allow to handle two accesses at the same time). The performance of a 5-3 processor with a dual-port memory is shown in Fig.11 (the utilization of the processor is the sum of utilizations of its 3 pipelines, so it ranges from 0 to 3).

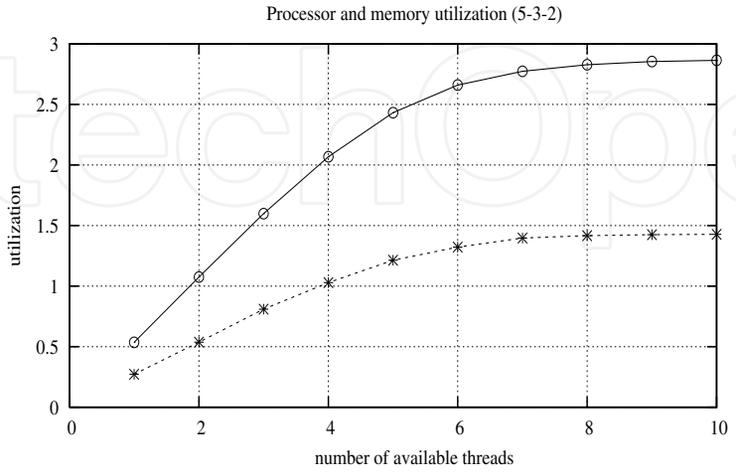


**Figure 11.** Processor (-o-) and memory (-x-) utilization for a 5-3 processor with dual–port memory; $l_t = 10$, $t_m = 5||2$, $t_{cs} = 1$, $p_{s1} = 0.2$, $p_{s2} = 0.1$

Fig.11 shows that for 3 pipelines and 5 simultaneous threads, the number of available threads greater than 6 provides the speedup that is almost equal to 3.

System bottlenecks can be identified by comparing service demands for different components of the system (in this case, the memory and the pipelines); the component with the maximum service demand is the bottleneck because it is the first component to reach its utilization limit and to prevent any increase of the overall performance. For a single runlength (of all simultaneous threads) the total service demand for memory is equal to $n_s * t_m$, while the service demand for each pipeline (assuming an ideal, uniform distribution of load over the pipelines) is equal to $n_s * \ell_t / n_p$. For a 4-2 processor, the service demands are equal (such a system is usually called "balanced"), so the utilizations of both, the processor and the memory, tend to their limits in a "synchronous" way. For a 5-3 processor with a dual-port memory, the service demand for the pipelines is greater than the service demand for memory, so the number of pipelines could be increased (by one pipeline); for more than 4 pipelines, the memory again becomes the bottleneck.

Simultaneous multithreading is quite flexible with respect to context switching times because the (simultaneous) threads fill the instruction issuing slots which normally would remain empty during context switching. Fig.12 shows the utilization of the processor and memory in a 1-1 processor with $t_{cs} = 3$, i.e., context switching time 3 times longer than in Fig.3. The reduction of the processor's utilization is more than 10 %, and is due to the additional 2 cycles of context switching which remain empty (out of 17 cycles, on average).

Fig.13 shows utilization of the processor and memory in a 2-1 processor, also for $t_{cs} = 3$. The reduction of utilization is much smaller in this case and is within 5 % (when compared with Fig.4).

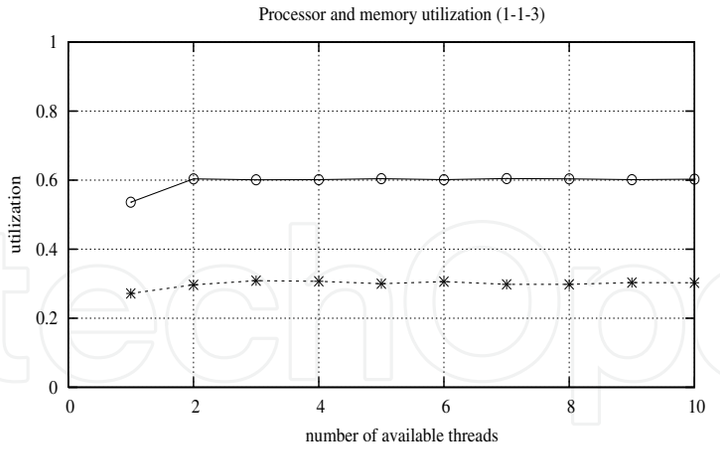


**Figure 12.** Processor (-o-) and memory (-x-) utilization for a 1-1 processor; $l_t = 10$, $t_m = 5$, $t_{cs} = 3$, $p_{s1} = 0.2$, $p_{s2} = 0.1$

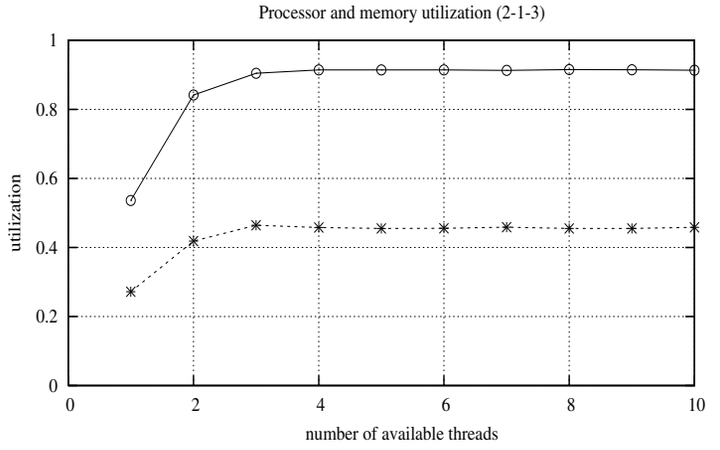


**Figure 13.** Processor (-o-) and memory (-x-) utilization for a 2-1 processor; $l_t = 10$, $t_m = 5$, $t_{cs} = 3$, $p_{s1} = 0.2$, $p_{s2} = 0.1$

## 5. Concluding remarks

Simultaneous multithreading discussed in this paper is used to increase the performance of processors by tolerating long–latency operations. Since the long–latency operations are playing increasingly important role in modern computer system, so is simultaneous multithreading. Its implementation as well as the required hardware resources are much simpler than in the case of out–of–order approach, and the resulting speedup scales well with the number of simultaneous threads. The main challenge of simultaneous multithreading is to balance the system by maintaining the right relationship between the number of simultaneous threads and the performance of the memory hierarchy.

All presented results indicate that the number of available threads, required for improved performance of the processor, is quite small, and is typically greater by 2 or 3 threads than the number of simultaneous threads. The results show that a larger number of available threads provides rather insignificant improvements of system's performance.

The presented models of multithreaded processors are quite simple, and for small values of modeling parameters ($n_t$, $n_p$, $n_s$) can be analyzed by the explorations of the state space. The following tables compare some results for the 1-1 processor and 3-2 processors:

| $n_t$ | *number of states* | *analytical utilization* | *simulated utilization* |
|---|---|---|---|
| 1 | 11 | 0.538 | 0.536 |
| 2 | 52 | 0.670 | 0.671 |
| 3 | 102 | 0.684 | 0.685 |
| 4 | 152 | 0.685 | 0.686 |
| 5 | 202 | 0.685 | 0.686 |

**Table 2.** A comparison of simulation and analytical results for 1-1 processors.

| $n_t$ | *number of states* | *analytical utilization* | *simulated utilization* |
|---|---|---|---|
| 1 | 11 | 0.538 | 0.536 |
| 2 | 80 | 1.030 | 1.031 |
| 3 | 264 | 1.384 | 1.381 |
| 4 | 555 | 1.568 | 1.568 |
| 5 | 951 | 1.655 | 1.647 |

**Table 3.** A comparison of simulation and analytical results for 3-2 processors.

The comparisons show that the results obtained by simulation of net models are very similar to the analytical results obtained from the analysis of states and state transitions.

A similar performance analysis of simultaneous multithreading, but using a slightly different model, was presented in [20]. All results presented there are very similar to results presented in this work which is an indication that the performance of simultaneous multithreaded systems is insensitive to (at least some) variations of implementation.

It should also be noted that the presented model is oversimplified with respect to the probabilities of pipeline stalls and does not take into account the dependence of stall probabilities on the history of instruction issuing. In fact, the model is "pessimistic" in this regard, and the predicted performance, presented in the paper, is worse than the expected performance of real systems. However, the simplification effects are not expected to be significant.

## Acknowledgement

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

## Author details

Wlodek M. Zuberek
*Memorial University, St.John's, Canada,*
*University of Life Sciences, Warsaw, Poland*

## 6. References


[1] Patterson, D.A., Hennessy, J.L. (2006). *Computer architecture – a quantitative approach* (4-th ed.); Morgan Kaufmann.

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

[3] Wilkes, M.V. (2001). "The memory gap and the future of high-performance memories"; *ACM Architecture News*, vol.29, no.1, pp.2-7.

[4] Sinharoy B. (1997). "Optimized thread creation for processor multithreading"; *The Computer Journal*, vol.40, no.6, pp.388-400.

[5] Baer, J-L. (2010). *Microprocessor architecture: from simple pipelines to chip multiprocessors*; Cambridge University Press.

[6] Jesshope, C. (2003). "Multithreaded microprocessors – evolution or revolution"; in *Advances in Computer Systems Architecture* (LNCS 2823), pp.21-45.

[7] Tseng, J. & Asanovic, K. (2003). "Banked multiport register files for high–frequency superscalar microprocessor"; *Proc. 30-th Int. Annual Symp. on Computer Architecture*, San Diego, CA, pp.62-71.

[8] Burger, D. & Goodman, J.R. (2004). "Billion–transistor architectures: there and back again"; *IEEE Computer*, vol.37, no.3, pp.22-28.

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

[10] Dennis, J.B. & Gao, G.R. (1994). "Multithreaded architectures: principles, projects, and issues"; in *Multithreaded Computer Architecture: a Summary of the State of the Art*, Kluwer Academic, pp.1-72.

[11] Ungerer, T., Robic, G. & Silc, J. (2002). "Multithreaded processors"; *The Computer Journal*, vol.43, no.3, pp.320-348.

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

[13] Mutlu, O., Stark, J., Wilkerson, C. & Patt, Y.N. (2003). "Runahead execution: an effective alternative to large instruction windows"; *IEEE Micro*, vol.23, no.6, pp.20-25.

[14] Zuberek, W.M. (1991). "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.

[15] Murata, T. (1989). "Petri nets: properties, analysis, and applications"; *Proceedings of the IEEE*, vol.77, no.4, pp.541-580.

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

[17] Zuberek, W.M. (1986). "M–timed Petri nets, priorities, preemptions, and performance evaluation of systems"; in *Advances in Petri Nets 1985* (LNCS 222), Springer-Verlag, pp.478-498.

[18] Zuberek, W.M. (1987). "D–timed Petri nets and modelling of timeouts and protocols"; *Transactions of the Society for Computer Simulation*, vol.4, no.4, pp.331-357.

[19] Zuberek, W.M. (1996). "Modeling using timed Petri nets – discrete–event simulation"; Technical Report #9602, Department of Computer Science, Memorial University, St. John's, Canada A1B 3X5.

[20] Zuberek, W.M. (2007). "Modeling and analysis of simultaneous multithreading"; *Proc. 14-th Int. Conf. on Analytical and Stochastic Modeling Techniques and Applications (ASMTA-07)*, a part of the *21-st European Conference on Modeling and Simulation (ECMS'07)*, Prague, Czech Republic, pp.115-120.