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), June 4-6, 2007, Prague, Czech Republic
Copyright ©2007 by European Council for Modeling and Simulation
ISBN 0-9553018-4-X, 978-0-9553018-4-1 (paper); 0-9553018-3-1, 978-0-9553018-3-4 (CDROM).

MODELING AND ANALYSIS OF SIMULTANEOUS MULTITHREADING

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

KEYWORDS
Simultaneous multithreading, instruction issuing, pipelined processors, timed Petri nets, performance analysis, event–driven simulation.

ABSTRACT
In simultaneous multithreading, several threads can issue instructions in each processor cycle. A simple and versatile timed Petri net model of simultaneous multithreading is proposed and is used to compare the performance of architectures with and without simultaneous multithreading. Performance results are obtained by event-driven simulation of net models and are verified by state–space–based analysis using combinations of modeling parameters for which the state space remains reasonably small.

INTRODUCTION

Continuous progress in manufacturing technologies results in the performance of microprocessors that has been steadily improving over the last decades, doubling every 18 months (the so called Moore's law [6]). At the same time, the capacity of memory chips has also been doubling every 18 months, but the performance has been improving less than 10% per year [10]. The latency gap between the processor and its memory doubles approximately every six years, and an increasing part of the processor's time is spent on waiting for the completion of memory operations [11]. Matching the performances of the processor and the memory is an increasingly difficult task. In effect, it is often the case that up to 60 % of execution cycles are spent waiting for the completion of memory accesses [8].

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 is to execute, during the 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 using the existing sequential instruction stream, it conveniently maintains code–base compatibility [7]. 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 [12]. Moreover, 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 [9], with a single 16 or 32-wide-issue processing core and huge branch predictors to sustain good instruction level parallelism, at present the industry is not moving toward the wide–issue superscalar model [1]. Design complexity and power efficiency direct the industry toward narrow–issue, high–frequency cores and multithreaded processors. According to [7]: "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 [3], [4] tolerates 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 [13].

In simultaneous multithreading [5], [7] several threads can issue instructions at the same time. If a processor contains more than one pipeline, or it contains several functional units, the instructions can be issued

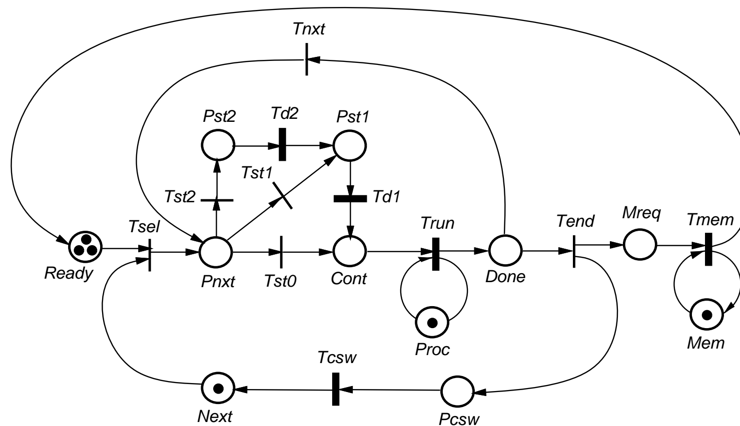

Fig.1. Petri net model of a multithreaded processor

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.

The main objective of this paper 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). 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 net model. Since the model is rather simple, simulation results can be verified (with respect to accuracy) by state–space–based performance analysis (for combinations of modeling parameters for which the state spaces remains reasonably small).

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*). *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 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. *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. 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, ℓ_t, i.e., the average number of instructions between two consecutive long–

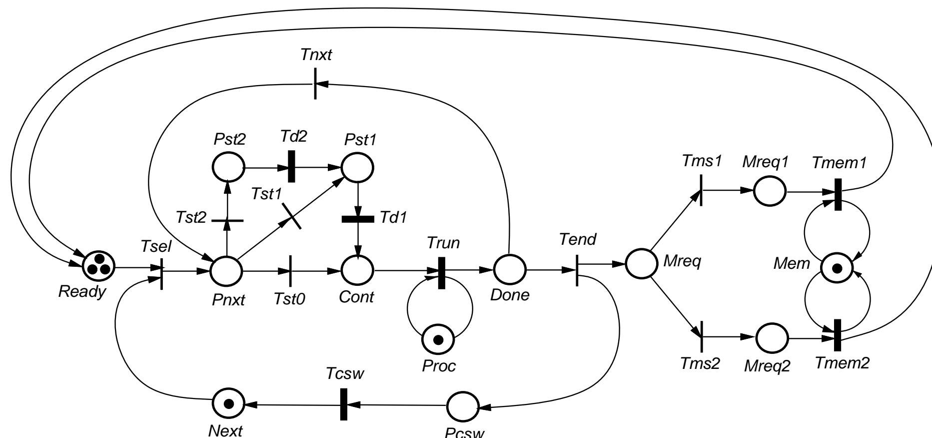

Fig.2. Petri net model of a multithreaded processor with a two–level memory

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 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 analyses), 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 summarized in Tab.1.

Table 1: Simultaneous multithreading modeling parameters and their typical values

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,...
ℓ_t	thread runlength	10
t_{cs}	context switching time	1,3
t_m	average memory access time	5
p_{s1}	prob. of one–cycle pipeline stall	0.2
p_{s2}	prob. of two–cycle pipeline stall	0.1

PERFORMANCE RESULTS

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.

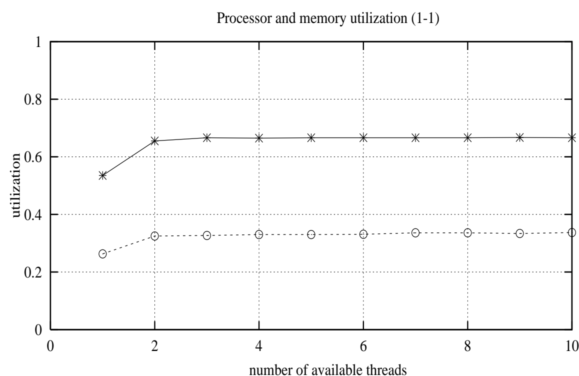

Fig.3. Processor (-x-) and memory (-o-) utilization for a 1-1 processor; $l_t = 10, t_m = 5, t_{cs} = 1$

The asymptotic value of the utilization can be estimated from the (average) number of empty 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 empty because of pipeline stalls. Moreover, there is an overhead of $t_{cs} = 1$ slot for context switching. The asymptotic utilization is thus 10/15 = 0.667, which corresponds very well with Fig.3.

The utilization of the processor can be improved by introducing a second (simultaneous) thread which issues its instructions in the unused slots. 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 1.00 (or 100 %).

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 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:

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

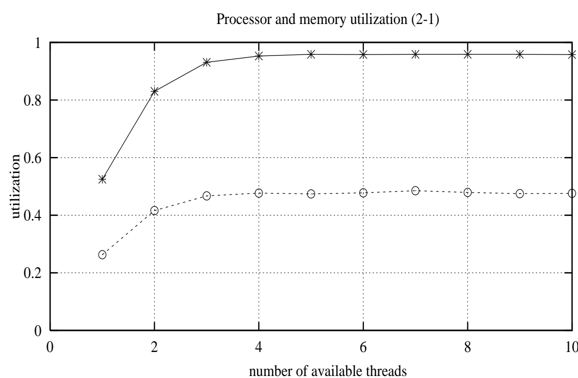

Fig.4. Processor (-x-) and memory (-o-) utilization for a 2-1 processor; $l_t = 10, t_m = 5, t_{cs} = 1$

The results for a 1-1 processor with a two–level memory are shown in Fig.5, and for a 2-1 processor in Fig.6.

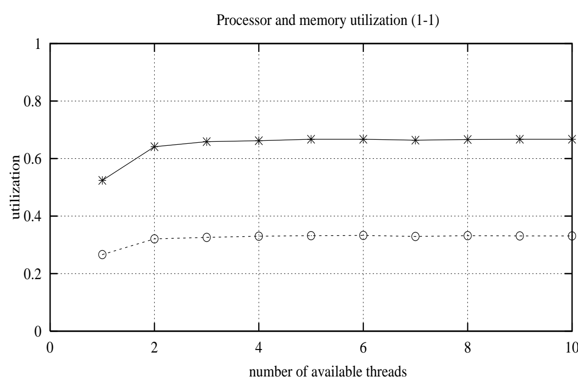

Fig.5. Processor (-x-) and memory (-o-) utilization for a 1-1 processor with 2-level memory; $l_t = 10$, $t_m = 4 + 20$, $t_{cs} = 1$

The results in Fig.5 and Fig.6 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.7.

When another instruction issue unit is added, the utilization increases by about 40 %, as shown in Fig.8.

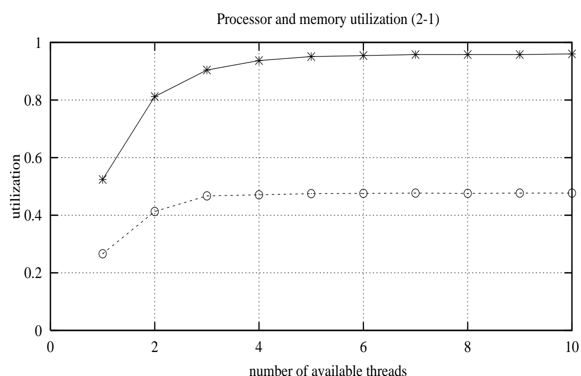

Fig.6. Processor (-x-) and memory (-o-) utilization for a 2-1 processor with 2-level memory; $l_t = 10$, $t_m = 4 + 20$, $t_{cs} = 1$

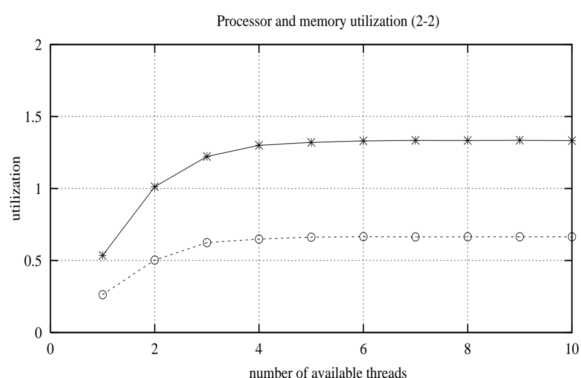

Fig.7. Processor (-x-) and memory (-o-) utilization for a 2-2 processor; $l_t = 10$, $t_m = 5$, $t_{cs} = 1$

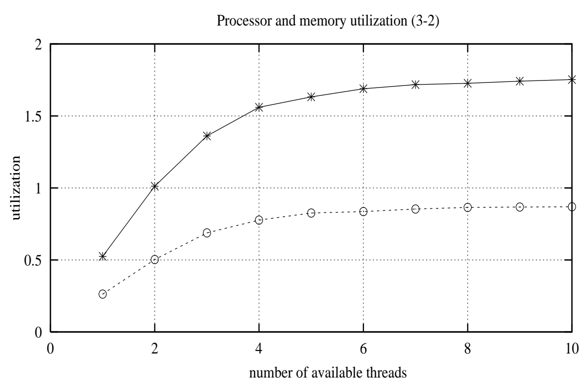

Fig.8. Processor (-x-) and memory (-o-) utilization for a 3-2 processor; $l_t = 10$, $t_m = 5$, $t_{cs} = 1$

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.9 (the utilization of the processor is the sum of utilizations of its 3 pipelines, so it ranges from 0 to 3).

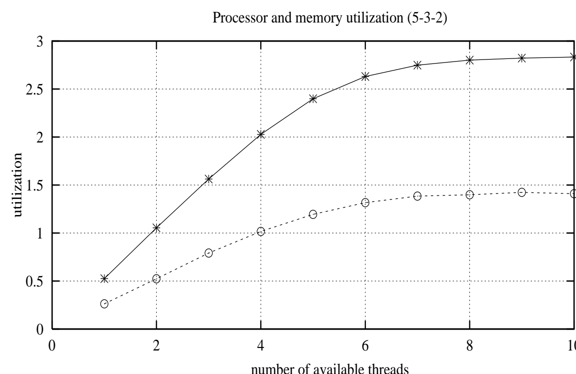

Fig.9. Processor (-x-) and memory (-o-) utilization for a 5-3 processor with dual–port memory; $l_t = 10$, $t_m = 5||2$, $t_{cs} = 1$

Fig.9 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 runlegth (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.

CONCLUDING REMARKS

Simultaneous multithreading discussed in this paper is a means to increase the performance of processors

by tolerating long–latency operations. Since the long–latency operations seem to be playing increasingly important role in modern microprocessors, 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. Performance improvement due to a larger number of available threads is rather insignificant.

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 table compares some results for the 1-1 processor:

n_t	*number of states*	*analytical utilization*	*simulated utilization*
1	11	0.526	0.535
2	57	0.656	0.655
3	107	0.666	0.666
4	157	0.666	0.666
5	207	0.667	0.666

For the 3-2 processor, the comparison is:

n_t	*number of states*	*analytical utilization*	*simulated utilization*
1	11	0.525	0.526
2	86	1.012	1.012
3	309	1.361	1.367
4	660	1.560	1.553
5	1,154	1.632	1.639

The results obtained by simulation of net models are very similar to the analytical results obtained from the analysis of states and state transitions. It should not be surprising that for more complex models the state space can become quite large, and then the event-driven simulation remains the best approach to analysis of net models.

ACKNOWLEDGEMENT

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

REFERENCES

[1] Burger, D., Goodman, J.R., “Billion–transistor architectures: there and back again”; IEEE Computer, vol.37, no.3, pp.22-28, 2004.

[2] Burger, D., Goodman, J.R., “Billion–transistor architectures”; IEEE Computer, vol.30, no.9, pp.46-49, 1997.

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

[4] Dennis, J.B., Gao, G.R., “Multithreaded architectures: principles, projects, and issues”; in “Multithreaded Computer Architecture: a Summary of the State of the Art”, pp.1-72, Kluwer Academic 1994.

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

[6] Hamilton, S., “Taking Moore’s law into the next century”; IEEE Computer, vol.32, no.1, pp.43-48, 1999.

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

[8] Mutlu, O., Stark, J., Wilkerson, C., Patt, Y.N., “Runahead execution: an effective alternative to large instruction windows”; IEEE Micro, vol.23, no.6, pp.20-25, 2003.

[9] Patt, Y.N., Patel, A., Friendly, D.H., Stark, J., “One billion transistors, one uniprocessors, one chip”; IEEE Computer, vol.30, no.9, pp.51-58, 1997.

[10] Patterson, D.A., Hennessy, J.L., “Computer architecture – a qualitative approach”, Morgan Kaufman 1996.

[11] Sinharoy B., “Optimized thread creation for processor multithreading”; The Computer Journal, vol.40, no.6, pp.388-400, 1997.

[12] Tseng, J., Asanovic, K., “Banked multiport register files for high–frequency superscalar microprocessor”; Proc. 30-th Int. Annual Symp. on Computer Architecture, pp.62-71, 2003.

[13] Ungerer, T., Robic, G., Silc, J., “Multithreaded processors”; The Computer Journal, vol.43, no.3, pp.320-348, 2002.

[14] 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.

BIOGRAPHICAL NOTE

W.M. ZUBEREK received M.Sc. degree in electronic engineering and Ph.D. and D.Sc. degrees in computer science, all from Warsaw University of Technology. Currently he is a Professor in the Department of Computer Science, Memorial University in St.John’s, Canada.