24-th IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems (FORTE'04), Performance Evaluation Worshop (EPEW'04), Toledo, Spain. Lecture Notes in Computer Science 3236, 209–219. © 2004 by Springer–Verlag.

Modeling and Analysis of Dual Block Multithreading [1]

Wlodek M Zuberek

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

Abstract. Instruction level multithreading is a technique for tolerating long–latency operations (e.g., cache misses) by switching the processor to another thread instead of waiting for the completion of a lengthy operation. In block multithreading, context switching occurs for each initiated long–latency operation. However, processor cycles during pipeline stalls as well as during context switching are not used in typical block multithreading, reducing the performance of a processor. Dual block multithreading introduces a second active thread which is used for instruction issuing whenever the original (main) thread becomes inactive. Dual block multithreading can be regarded as a simple and specialized case of simultaneous multithreading when two (simultaneous) threads are used to issue instructions for a single pipeline. The paper develops a simple timed Petri net model of a dual block multithreading and uses this model to estimate the performance improvements of the proposed dual block multithreading.

Keywords: Block multithreading, instruction issuing, pipelined processors, timed Petri nets, performance analysis, event–driven simulation.

1 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 [4]). 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 [5]. 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 [8]. Matching the performances of the processor and the memory is an increasingly difficult task [9].

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 [6]. In effect, the instruction stream is dynamically decomposed into

[1] This version of the paper is slightly different from the original one. The paper has been revised by making several straightforward corrections and improvements.

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 [11]. Moreover, memory latencies are so long that out–of–order processors require very large instruction windows to tolerate them. A cache miss to main memory costs about 128 cycles on Alpha 21264 [13] and 330 cycles on a Pentium-4–like processor [10]. Large instruction windows mean design complexity, verification difficulty and increased power consumption [7], so the industry is not moving toward the wide–issue superscalar model [1]. In effect, it is often the case that up to 60 % of execution cycles are spent waiting for the completion of memory accesses [7].

Instruction–level multithreading [2], [3] 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 [12].

In block multithreaded processors, the pipeline is stalled occasionally for one or more processor cycles because of the instruction dependencies. Since the trend in modern microprocessors is to increase the depth of the pipelines [10], and deep pipelines increase the probability of pipeline stalls due to instruction dependencies, the effects of pipeline stalls on the performance of processors can be quite significant. This paper proposes a variant of block multithreading in which an additional active thread is used to issue instruction in those processor cycles in which the main thread is inactive. The proposed approach is called dual block multithreading.

The main objective of this paper is to study the performance of dual block multithreaded processors in order to determine how effective the addition of the second active thread can be. 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. 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).

2 Petri Net Models

A timed Petri net model of a simple block multithreaded processor at the instruction execution level 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*.

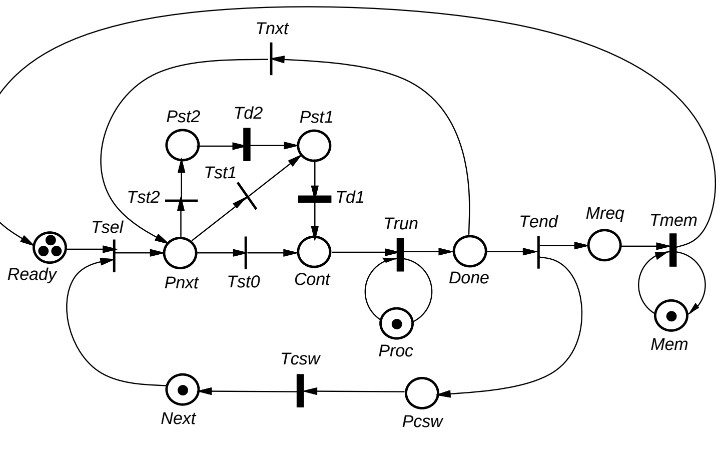

Fig.1. Petri net model of a block multithreaded processor.

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–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.

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*. For a model of dual block multithreading, the initial marking of *Next* is 2.

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, 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*).

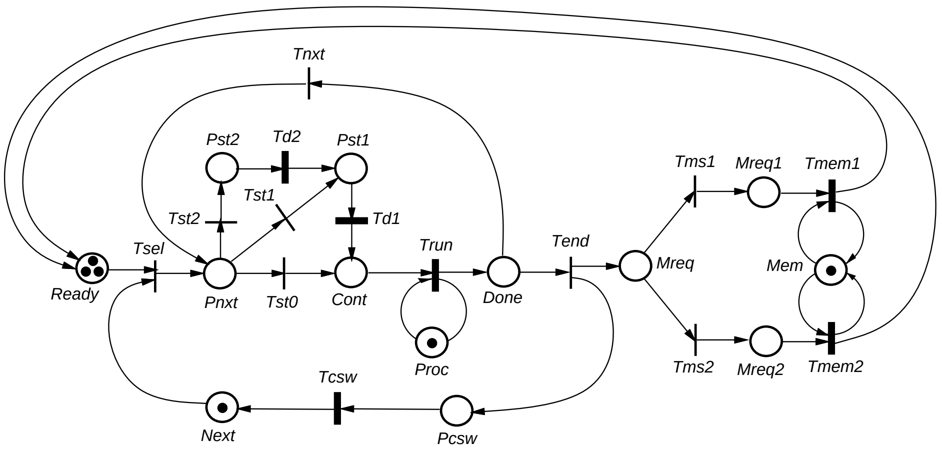

Fig.2. Petri net model of a block multithreaded processor with a two–level memory.

The effects of memory hierarchy can easily 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).

For convenience, all temporal properties of the model 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 context switching (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. The number of available threads, n_t, changes from 1 to 10 in order to check if a large number of threads has can provide a reasonable improvement of the processor's performance. Thread runlength, ℓ_t, equal to 10 corresponds to the (primary) cache miss of 10%. Context switching times equal to 1 and 5 are used to check the sensitivity of performance results on the duration of context switching. The average memory access time, t_m, of 10 processor cycles matches the thread runlength, ℓ_t, providing the balanced utilization of the processor and the memory; if $t_m > \ell_t$, the memory becomes the bottleneck which limits the performance of the system; if $\ell_t > t_m$, the memory has little influence on the system's performance. The probabilities of pipeline stalls, p_{s1} and p_{s2}, correspond to the probabilities of data hazards used in [5].

Table 1. Block multithreading modeling parameters and their typical values.

symbol	*parameter*	*value*
n_t	number of available threads	1,...,10
ℓ_t	thread runlength	10
t_{cs}	context switching time	1,5
t_m	average memory access time	10
p_{s1}	prob. of one–cycle pipeline stall	0.2
p_{s2}	prob. of two–cycle pipeline stall	0.1

3 Performance Results

The utilization of the processor, as a function of the number of available threads, for a "standard" processor (i.e., a processor with a single instruction issue unit) is shown in Fig.3.

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 a dual block multithreaded processor, i.e., a processor issuing instructions to a single instruction execution pipeline from two (simultaneous) threads.

The utilization of the processor is improved by about 40 %.

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

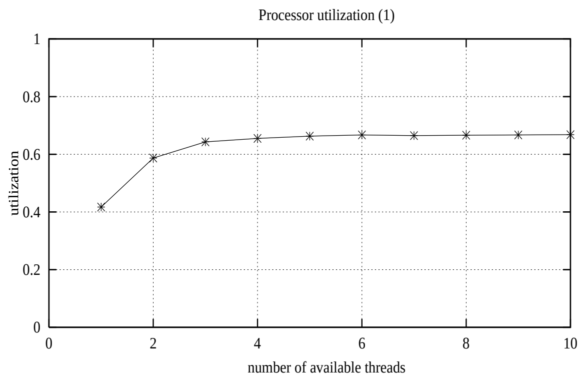

Fig.3. Processor utilization for standard block multithreading; $l_t = 10$, $t_m = 10$, $t_{cs} = 1$, $p_{s1} = 0.2$, $p_{s2} = 0.1$.

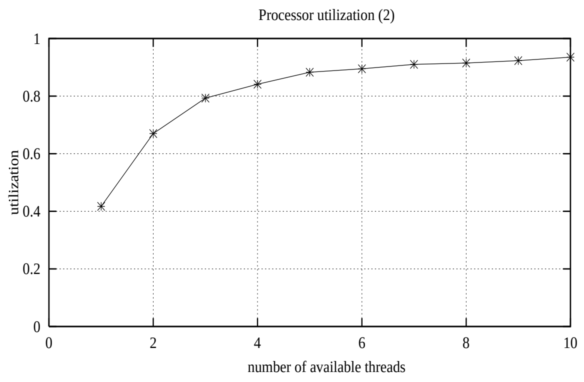

Fig.4. Processor utilization for dual block multithreading; $l_t = 10$, $t_m = 10$, $t_{cs} = 1$, $p_{s1} = 0.2$, $p_{s2} = 0.1$.

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 $t_m = 10$). Let the two levels of memory have access times equal to 8 and 40, 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:

$$8 * \frac{15}{16} + 40 * \frac{1}{16} \ = \ 10.$$

The results for a standard block multithreaded processor with a two–level memory are shown in Fig.5, and for a dual block multithreaded processor in Fig.6.

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-

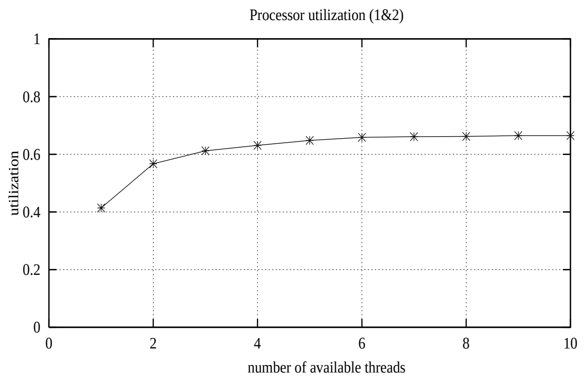

Fig.5. Processor utilization for standard block multithreading with 2-level memory; $l_t = 10$, $t_m = 8 + 40$, $t_{cs} = 1$, $p_{s1} = 0.2$, $p_{s2} = 0.1$.

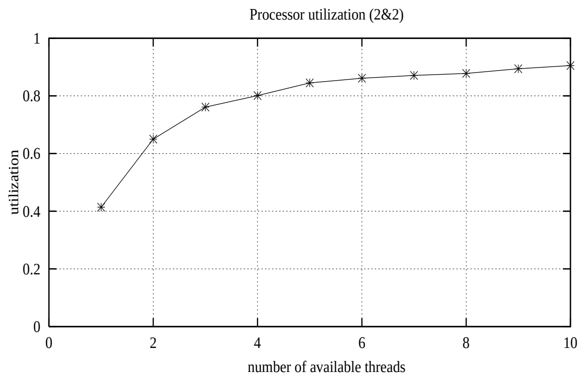

Fig.6. Processor utilization for dual block multithreading with 2-level memory; $l_t = 10$, $t_m = 8 + 40$, $t_{cs} = 1$, $p_{s1} = 0.2$, $p_{s2} = 0.1$.

level memory models; the multiple levels of memory hierarchy apparently have no significant effect on the performance results.

Dual multithreading is also quite flexible with respect to context switching times because the additional thread fills the instruction issuing slots which normally would remain empty during context switching. Fig.7 compares the utilization of the standard block multithreaded processor with $t_{cs} = 1$ (broken line) and $t_{cs} = 5$ (solid line). The reduction of the processor's utilization for $t_{cs} = 5$ is about 20 %, and is due to the additional 4 cycles of context switching which remain empty (out of 19 cycles, on average).

Fig.8 compares utilization of the dual block multithreaded processor for $t_{cs} = 1$ and $t_{cs} = 5$. The reduction of utilization is much smaller in this case and is within 10 %.

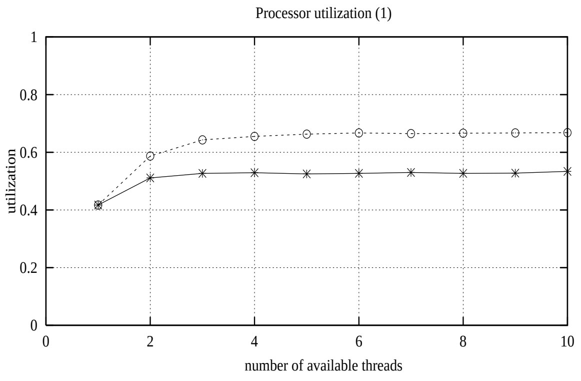

Fig.7. Processor utilization for standard block multithreading; $l_t = 10$, $t_m = 10$, $t_{cs} = 1, 5$, $p_{s1} = 0.2$, $p_{s2} = 0.1$.

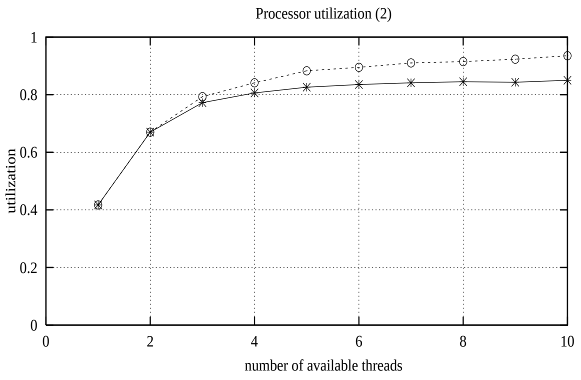

Fig.8. Processor utilization for dual block multithreading; $l_t = 10$, $t_m = 10$, $t_{cs} = 1, 5$, $p_{s1} = 0.2$, $p_{s2} = 0.1$.

4 Concluding Remarks

Dual block multithreading discussed in this paper increases the performance of processors by tolerating long–latency operations (block multithreading) and pipeline stalls (dual multithreading). Its implementation is rather straightforward while the improvement of the utilization of processors can be quite significant, as shown in Fig.9.

However, the improved performance of dual multithreading can be obtained only if the system is balanced, or if the processor is the system's bottleneck. Fig.10 shows the utilization of the processor for standard (solid line) as well as dual multithreading (broken line); the utilizations of both processors are practically identical because, in these particular cases, the memory is the system's bottleneck that restricts the performance of other components.

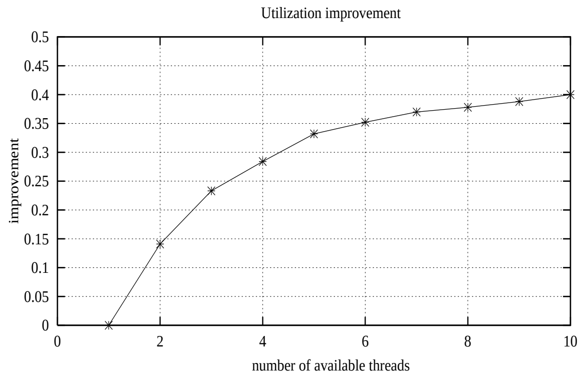

Fig.9. The improvement of processor utilization due to dual block multithreading; $l_t = 10$, $t_m = 10$, $t_{cs} = 1$, $p_{s1} = 0.2$, $p_{s2} = 0.1$.

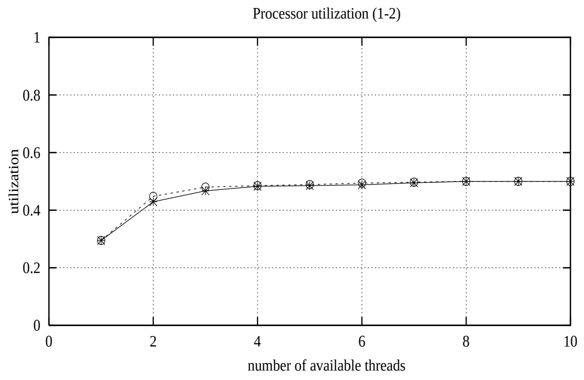

Fig.10. A comparison of processor utilization; $l_t = 10$, $t_m = 20$, $t_{cs} = 1$, $p_{s1} = 0.2$, $p_{s2} = 0.1$.

All presented results indicate that the number of available threads, required for improved performance of the processor, is rather small, and is practically not greater than 4 threads. Performance improvement due to a larger number of available threads is rather insignificant.

Obtained processor utilization results are consistent with other studies of the performance of multithreaded architectures [16], [15]. The performance of distributed memory multithreaded multiprocessor systems can be compared with the results presented in this paper by assuming that the probability of accessing local nodes is equal to 1 (which means that the nodes can be analyzed in isolation).

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 standard block multithreaded processor:

n_t	*number of states*	*analytical utilization*	*simulated utilization*
1	11	0.417	0.417
2	107	0.591	0.587
3	207	0.642	0.643
4	307	0.658	0.655
5	407	0.664	0.663

For a dual block multithreaded processor the comparison is:

n_t	*number of states*	*analytical utilization*	*simulated utilization*
1	11	0.417	0.417
2	130	0.672	0.670
3	320	0.793	0.793
4	642	0.848	0.841
5	972	0.878	0.883

The simulation–based results shown in the tables 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. For example, the state space for the dual multithreaded processor increases by more than 300 states for each additional thread (above 3). Analytical solution of very large systems of linear equations (which describe the stationary probabilities of states) may require special numerical techniques to provide the necessary accuracy. Therefore, discrete–event simulation of net models is an attractive alternative to exhaustive state space exploration of complex models.

Finally, it should 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 of real systems. On the other hand, the simplicity of the presented model is likely to outweight its simplification(s) as the simplification effects are not expected to be significant.

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

4. Hamilton, S., “Taking Moore’s law into the next century”; IEEE Computer, vol.32, no.1, pp.43-48, 1999.
5. Hennessy, J.L., Patterson, D.A., “Computer architecture – a qualitative approach” (3 ed.), Morgan Kaufman 2003.
6. Jesshope, C., “Multithreaded microprocessors – evolution or revolution”; in “Advances in Computer Systems Architecture” (LNCS 2823), pp.21-45, 2003.
7. 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.
8. Sinharoy B., “Optimized thread creation for processor multithreading”; The Computer Journal, vol.40, no.6, pp.388-400, 1997.
9. Sohi, G.S., “Microprocessors – 10 years back, 10 years ahead”; in “Informatics: 10 Years Back, 10 Years Ahead” (Lecture Notes in Computer Science 2000), pp.209-218, 2001.
10. Sprangle, E., Carmean, D., “Increasing processor performance by implementing deeper pipelines”; Proc. 29-th Annual Int. Symp. on Computer Architecture, Anchorage, Alaska, pp.25-34, 2002.
11. 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.
12. Ungerer, T., Robic, G., Silc, J., “Multithreaded processors”; The Computer Journal, vol.43, no.3, pp.320-348, 2002.
13. Wilkes, M.V., “The memory gap and the future of high-performance memories”; ACM Architecture News, vol.29, no.1, pp.2-7, 2001.
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.
15. Zuberek, W.M., “Analysis of pipeline stall effects in block multithreaded multiprocessors”; Proc. 16-th Performance Engineering Workshop, Durham, UK, pp.187-198, 2000.
16. Zuberek, W.M., “Analysis of performance bottlenecks in multithreaded multiprocessor systems”; Fundamenta Informaticae, vol.50, no.2, pp.223-241, 2002.