IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC 1998); San Diego, California, 11-14 October 1998, pp-232-237.

Performance Bounds for Distributed Memory Multithreaded Architectures

W.M. Zuberek

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

R. Govindarajan

Supercomputer Education and Research
Center, Indian Institute of Science
Bangalore 560 012, India

Abstract

In distributed memory multithreaded systems, the long memory latencies and unpredictable synchronization delays are tolerated by context switching, i.e., by suspending the current thread and switching the processor to another thread waiting for execution. Simple analytical upper bounds on performance measures are derived using throughput analysis and extreme values of some model parameters. These derived bounds are compared with performance results obtained by simulation of a detailed model of the analyzed architecture.

1. INTRODUCTION

Multithreading provides a means of tolerating long, unpredictable communication latency and synchronization delays in distributed memory multiprocessor systems. Its basic idea is quite straightforward. In a traditional architecture, when a processor accesses a location in memory, it waits for the result, possibly after executing a few instructions that are independent of the memory operation. In a large multiprocessor, this wait may involve more than one hundred processor cycles [6] since the memory request may need to be transmitted across the communication network to a remote memory module, serviced, and then the result returned. Not surprisingly, the utilization of processors in such systems tends to be low. Alternatively, if the processor maintains multiple threads of execution, and can quickly switch from one thread to another, instead of waiting for completion of memory requests, the processor can switch to another thread and continue is execution. Thus the processor can overlap some computation during memory access, effectively 'hiding' the large memory latency of distributed memory architectures. With multithreading, the processor utilization can be largely independent of the latency in completing remote memory accesses.

Traditional multithreaded architectures [1, 4, 6, 16] *issue* instructions from only one thread each cycle. The available instruction-level parallelism exploited by a single thread in the multithreaded execution is limited. Consequently, the utilization of processor resources is lower for multithreaded execution. To exploit greater instruction-level parallelism supported by modern superscalar processors which feature aggressive multiple instruction issue and execution, recent multithreaded architectures propose to extract instruction-level parallelism by grouping instructions from multiple instruction streams or threads. We generically refer to these architectures as simultaneous multithreading. Several simultaneous multithreaded architectures have been reported in the literature [12, 10, 8, 17].

Designing a multithreaded processor is an intricate process since each design decision impacts upon others. For example, a single–instruction thread size may be desirable (e.g., HEP [16]) because instruction dependencies in pipelined execution can be eliminated (consecutive instruction are fetched from different threads). However, such a small thread size forces at least one scheduling decision per CPU clock cycle. Since careful allocation of the CPU resource is vital for efficient execution of many applications, a larger thread size has to be tolerated so that suitable scheduling decision can be made.

Several multithreaded architectures have been proposed which differ in the implementation of multithreading [1, 4, 5, 6, 8, 10, 12]. They differ in two basic aspects, in the number of instructions executed before switching to another thread (one, several, as many as possible), and the cause of context switching (every load, remote load).

It is assumed in this paper that context switching can be performed very efficiently (in one processor cycle); consequently, context switching on every load is considered in greater detail. Other approaches to context switching require some modifications of the proposed approach.

An interconnection network links the nodes of distributed memory multiprocessor architecture. Interconnection networks can have different topologies and different properties depending upon the topology. It is assumed that all messages in the system are routed along the shortest paths, but in a non–deterministic

manner. That is, whenever there are multiple (shortest) paths between the source and destination, any of the paths is equally likely to be used. The delay of each message is proportional to the number of hops between the source and destination nodes, and it also depends upon the traffic in the chosen path. The interface between the network switch and processor node is through a pair of outbound and inbound network interfaces.

Analyzing the performance of such architectures is rather involved as it depends on a number of parameters related to the architecture — memory latency time, context switching time, switch delay in the interconnection network — and a number of application parameters — number of parallel threads, runlengths of threads, remote memory access pattern and so on. A number of analytical performance evaluation studies of multithreaded architectures have been reported in the literature [2, 3, 14, 11, 15, 18].

The steady–state behavior of systems can be characterized by 'flows' of activities (such as requests for service and results of these requests) between different components of the system. The basic property of the steady–state is that these flows must be balanced, i.e., the total flow incoming into each component must be equal to the total flow outgoing from it. This balance of flows can be used for analysis of throughputs [7] of system's components. Intuitively, the throughput of a component is equal to the average number of service requests (or service completions) in a unit of time. Utilizations of components can be derived very easily from throughputs [7]. Some other performance measures can also be obtained by applying general rules of operational analysis [13].

This paper describes simple models of distributed memory multithreaded architectures, derives upper bounds on some performance results, and and compares these bounds with results obtained by simulation of a detailed model. The influence of architectural and application parameters on the performance of the system is also discussed.

2. MODEL OF THE SYSTEM

In distributed memory architectures, each node has local (non–shared) memory, and is connected to a number of neighboring nodes, usually in a regular pattern. A two–dimensional torus network for a 16–processor systems is shown in Fig.2.1; such a 16–processor system is used as a running example in this paper, but the results can easily be extended to higher number of processors and other types of interconnection networks.

Each node in this system contains a (multithreaded) processor and two switches which connect the processor with the network and forward the messages to/from the four adjacent nodes (one switch handles node outgoing traffic, and the other handles incoming messages as well as those messages that are forwarded to other nodes).

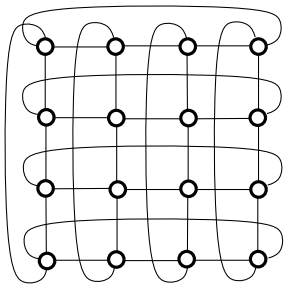

Fig.2.1. An outline of a 16–processor system.

In the multithreaded execution model, a program is a collection of partially ordered threads, and a thread consists of a sequence of instructions which are executed in the conventional von Neumann model. It is assumed that switching from one tread to another is performed on every load [4, 9]. If the currently executed instruction issues an operation of accessing either a local or a remote memory location, the execution of the current thread suspends, and another ready thread is selected from a pool of ready threads. When the long–latency operation for which the thread was suspended is finished, the thread becomes ready and joins the pool of threads waiting for execution.

The average number of instructions executed by a thread before issuing a load operation (and switching to another thread) is called *thread runlength*, and is one of important model parameters.

Fig.2.2 outlines the architecture of a single processor. The ready threads wait for execution in the *Ready Queue*. When the thread executing in the *Processor* issues a (long–latency) memory operation, the thread becomes suspended, context switching initiates execution of another thread from the *Ready Queue*, and the memory operation request is directed to either local or remote memory. For local memory operations, the request enters the *Memory Queue*, and, when the memory becomes available, is serviced, after which the suspended thread becomes ready and joins the *Ready Queue*.

For remote memory accesses, the request joins the *Outbound Queue* and is forwarded (by the *Outbound Switch*) to one of neighboring nodes. It is assumed that all messages in the system are routed to their destinations along the shortest paths, and whenever there are multiple (shortest) paths between the source and destination, any one of these paths is equally likely to be taken (so the traffic is distributed uniformly in the interconnecting network).

Requests (or messages) sent into the interconnecting network enter nodes through *Inbound Queue* and *Inbound Switch*. If the request is to be forwarded to yet another node (another 'hop'), it is sent from the *Inbound Switch* directly to another node. If the request has reached its target node, it enters the *Mem-*

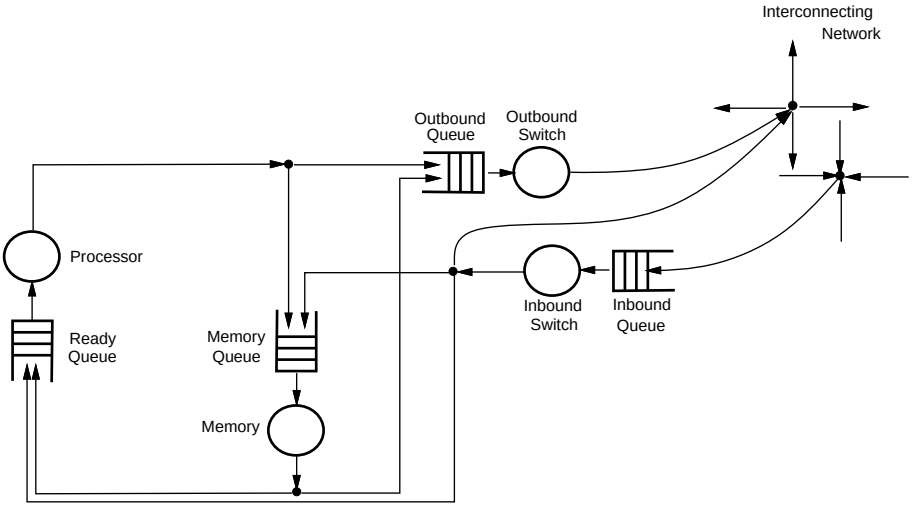

Fig.2.2. Model of a single multithreaded processor.

ory Queue, and after accessing the *Memory*, the result is sent back to the request's source through *Outbound Queue* and *Outbound Switch*. When the result returns to the source node, it changes the status of the suspended thread to ready, and the thread joins the *Ready Queue*.

The most important parameters which affect the performance of the architecture shown in Fig.2.2 include thread runlength (or the average service time of *Processor*), the *Memory* cycle time (i.e., the service time of *Memory*), the delay of *Inbound Switch* and *Outbound Switch*, and the probability of local (or remote) memory accesses (which somehow characterizes the 'locality' of memory references).

It is assumed that thread execution time is exponentially distributed with the mean value equal to the thread runlength. All other service times (for *Memory*, *Inbound Switch* and *Outbound Switch* servers) are constant. The context switching time is not represented explicitly but is included in the thread runlength (it appears that this simplifies the model without affecting the performance results in any significant way [9]). Similarly, a geometric distribution of the execution time (with the same mean value) would be a more realistic representation of the runlength than the exponential one, as it would represent the execution of consecutive instructions of the thread. However, the model with exponential distribution is simpler, and was found to be practically as accurate as the one with geometric distribution [9].

The delay of remote memory accesses is proportional to the number of hops in the interconnecting network; it also depends upon the traffic in the chosen path. The average number of hops can be estimated assuming that the accesses are uniformly distributed over the nodes of the system. For a 16–processor system with a torus–like network, the average number of hops, n_h, is approximately equal to 2 [9].

It should be observed that the original distribution of the number of hops (as discussed above) is not preserved in the model shown in Fig.2.2. In order to make the model simple, the geometric distribution of the number of hops is used with the same mean value as the original distribution (i.e., 2 for the 16–processor system). However, simulation results indicate that this simplification affects the results only insignificantly [9].

The main parameters used in the model are:

parameter	*symbol*
thread runlength	ℓ_t
processor cycle time	t_p
memory cycle time	t_m
switch delay	t_s
average number of hops (one direction)	n_h
probability to access local memory	p_ℓ
probability to access remote memory	p_r

In most cases, only relative values of temporal parameters (e.g., the processors cycle time, the switch delay) are needed, so it is convenient to express the values of other parameters in terms of a selected one; it is assumed that the processor cycle time is one unit, and all other temporal parameters are relative to the processor cycle time. For example, the memory cycle time is 10 (processor cycle times), and the switch delays are 5 or 10 (again, processor cycle times).

3. PERFORMANCE BOUNDS

If the probability of accessing local memory, p_ℓ, is close to 1.0, the model can be simplified by neglecting the interconnection network and its influence; in fact, in such a case each node can be considered in isolation, as shown in Fig.3.1.

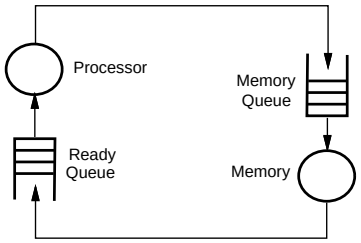

Fig.3.1. Simplified model for local memory accesses.

For this simple cyclic model, if the number of threads, n_t, is small, there is practically no queueing, so the throughput (the same for processor and the memory in this case) can be expressed as:

$$\frac{n_t}{\ell_t + t_m}$$

so the utilization of the processor is:

$$u'_p = n_t * \frac{\ell_t}{\ell_t + t_m}$$

and the utilization of the memory is:

$$u'_m = n_t * \frac{t_m}{\ell_t + t_m}$$

(the utilization of switches is zero in this case).

If the number of threads is large, the system enters its saturation, and the throughput is limited by the element with the maximum service demand (the so called bottleneck). Since the visit rates for processor and memory are the same (both equal to 1), the bottleneck is determined by the (average) service time of elements; if $\ell_t > t_m$, the processor is the bottleneck, its utilization is close to 100%, and the utilization of memory is:

$$u''_m = \frac{t_m}{\ell_t}.$$

On the other hand, if $\ell_t < t_m$, the memory is the bottleneck, its queue contains most of the requests waiting for service, memory utilization is close to 100%, while the utilization of the processor is:

$$u''_p = \frac{\ell_t}{t_m}.$$

If $\ell_t = t_m$, both the processor and memory are utilized approximately 100%.

If the probability of accessing local memory, p_ℓ, is close to 0.0, only remote memory accesses should be considered, and then the simplified model can be as shown in Fig.3.2.

A very straightforward expansion of the loops on the inbound switches, taking into account that the average

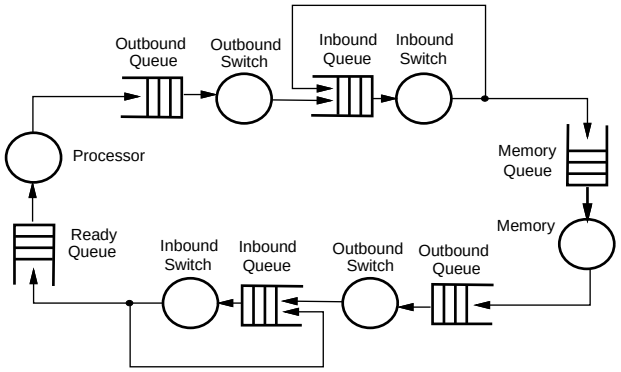

Fig.3.2. Simplified model for remote memory accesses.

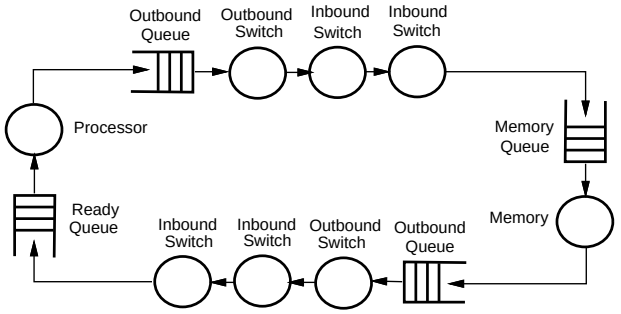

Fig.3.3. Expanded model for remote memory accesses.

number of hops (for a 16–processor system) is equal to 2, results in the model shown in Fig.3.3.

If the number of threads, n_t, is small, the queueing can be ignored, so the throughput is:

$$\frac{n_t}{\ell_t + t_m + 2 * (1 + n_h) * t_s}$$

and then the utilization of the processor is:

$$u'_p = n_t * \frac{\ell_t}{\ell_t + t_m + 2 * (1 + n_h) * t_s}$$

while the utilization of the memory is:

$$u'_m = n_t * \frac{t_m}{\ell_t + t_m + 2 * (1 + n_h) * t_s}$$

If the number of threads, n_t, is sufficiently large, the system enters the saturation region, in which the performance of the whole system is limited by the bottleneck. As before, the bottleneck is the system's component with the maximum service demand. In Fig.3.3, the visit ratios for the inbound and outbound switches are 4 and 2, respectively (in general case, these visit ratios are $2 * n_h$ and 2 for inbound and outbound switches, respectively).

The three possibilities for a bottleneck are:

- the inbound switch (if $2 * n_h * t_s > \max(\ell_t, t_m)$): the utilizations of the processor and the memory are $\ell_t/(2*n_h*t_s)$ and $t_m/(2*n_h*t_s)$, respectively;
- the processor (if $\ell_t > \max(2*n_h*t_s, t_m)$): the utilizations of the memory and the (inbound) switch are t_m/ℓ_t and $2*n_h*t_s/\ell_t$, respectively;
- the memory (if $t_m > \max(\ell_t, 2*n_h*t_s)$): the utilizations of the processor and the (inbound) switch are ℓ_t/t_m and t_s/t_m, respectively.

An illustration of these bounds is shown in Fig.3.4 for local and remote memory accesses. Let $\ell_t = t_m = 10$, $t_s = 5$, and $n_h = 2$. If the number of threads is small and $p_\ell \to 1$, the bound on processor utilization is $0.5 * n_t$, which is represented (in Fig.3.4) by a straight line with slope 0.5; if $p_\ell = 0$, the bound is a straight line with slope 0.2. If the number of threads, n_t, is large, the bound on processor utilization is constant (i.e., does not depend upon the number of threads, n_t); this bound is at the level of 1 for $p_\ell = 1$, and at the level of 0.5 for $p_\ell = 0$.

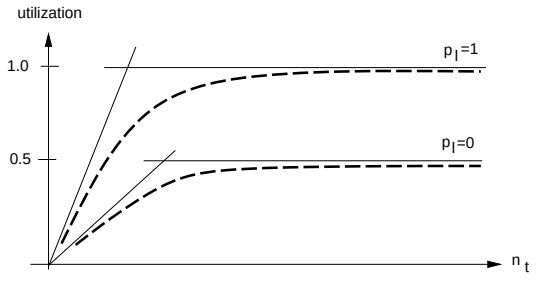

Fig.3.4. Example performance bounds.

The two hypothetical utilization curves (as function of n_t), for $p_\ell = 0$ and for $p_\ell = 1$, are shown in Fig.3.4 by dashed lines.

The derived performance bounds can be compared with the results obtained by simulation of the detailed model of distributed memory multiprocessor architecture [9, 18]. Fig.3.5 shows the utilization of the processor as a function of two variables, the probability of accessing local memory, p_ℓ, and the number of threads, n_t, for $\ell_t = t_m = t_s = 10$.

The derived upper bounds on processor utilization are as follows:

case	*bound*
$p_\ell = 1, n_t \to 0$	$0.5 * n_t$
$p_\ell = 1, n_t \to \infty$	1.0
$p_\ell = 0, n_t \to 0$	$0.125 * n_t$
$p_\ell = 0, n_t \to \infty$	0.25

For small values of p_ℓ and large numbers of threads, the processor's utilization is limited by the inbound switch, which is the bottleneck ($2*n_h*t_s > \max(\ell_t, t_m)$).

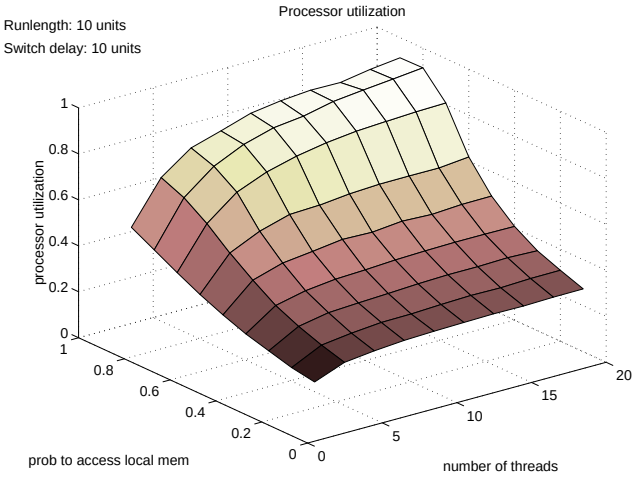

Fig.3.5. Processor utilization ($t_s = 10$).

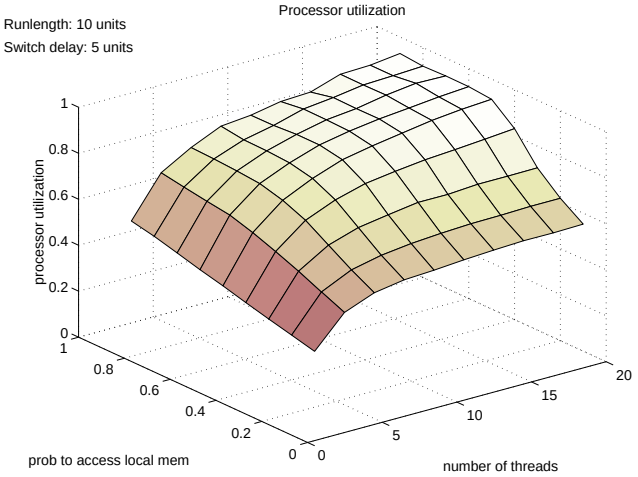

Fig.3.6. Processor utilization ($t_s = 5$).

Fig.3.6 also shows the processor utilization as a function of p_ℓ and n_t, but in this case the delay of switches is 5 while $\ell_t = m_t = 10$. For this case, the derived upper bounds on processor utilization are:

case	*bound*
$p_\ell = 1, n_t \to 0$	$0.5 * n_t$
$p_\ell = 1, n_t \to \infty$	1.0
$p_\ell = 0, n_t \to 0$	$0.2 * n_t$
$p_\ell = 0, n_t \to \infty$	0.5

Fig.3.5 and Fig.3.6 show that the derived bounds are quite tight. Moreover, the utilization surface shown in Fig.3.6 contains a characteristic 'ridge' at $p_\ell = 0.5$; it appears [18] that $p_\ell = 0.5$ is the condition at which the bottleneck in this system changes – for $p_\ell < 0.5$, the inbound switch is the bottleneck, so the processor utilization is actually limited by the throughput of the inbound switch; for $p_\ell > 0.5$, the processor becomes the bottleneck, and its utilization (for large numbers of

threads) is close to 100%. The same effect occurs in Fig.3.5 at $p_\ell = 0.75$ [18].

4. CONCLUDING REMARKS

Simple formulas for upper bounds on utilization of the components of distributed memory multithreaded architectures are obtained by throughput analysis for extreme values of some model parameters. The bounds are verified by comparison with results obtained by simulation of a detailed model. The derived bounds can easily be generalized. For example, it should be noticed that for small number of threads (when queueing can be neglected), the bound on the utilization, for any value of p_ℓ between 0 and 1, can be obtained as a linear combination of bounds for $p_\ell = 0$ and $p_\ell = 1$; this is clearly illustrated in Fig.3.5 and Fig.3.6 by a practically straight–line boundary of the utilization surface along the p_ℓ axis.

The proposed approach can easily be adapted to other models of multithreading. For example, context switching on remote loads (typically used in systems with 'slow' context switching), requires only a few modifications of the presented approach (in fact, the bounds for $p_\ell = 0$ remain the same; the bounds for $p_\ell = 1$ need to be revised).

Finally, the upper bounds on component utilizations must be invariant under performance–preserving model transformations. An example of model simplification was proposed in [9], and it can easily be checked that the proposed simplification preserves the upper bounds discussed in this paper. Consequently, the simplified model may provide slightly different results for some combinations of model parameters, but its asymptotic behavior is consistent with the original model.

References

[1] 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.

[2] Agrawal, A., "Limits on interconnection network performance"; IEEE Trans. on Parallel and Distributed Systems, vol.2, no.4, pp.398-412, 1991.

[3] Alkalaj, L., Boppana, R.V., "Performance of a multithreaded execution in a shared–memory multiprocessor"; Proc. 3-rd Annual IEEE Symp. on Parallel and Distributed Processing, Dallas, TX, pp.330-333, 1991.

[4] 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.

[5] Boothe, B. and 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] Culler, D.E., et al., "Fine–grain parallelism with minimal hardware support: a compiler controlled threaded abstract machine"; Proc. 4-th Int. Conf. on Architectural Support of Programming Languages and Operating Systems, Santa Clara, CA, pp.164-175, 1991.

[7] Ferrari, D., "Computer systems performance evaluation"; Prentice–Hall 1978.

[8] 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.

[9] 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, 1997.

[10] Hirata, H., et al., "An elementary processor architecture with simultaneous instruction issuing from multiple threads"; Proc. 19-th Int. Symp. on Computer Architecture, pp.136-145, 1992.

[11] Johnson, K., "The impact of communication locality on large–scale multiprocessor performance"; Proc. 19-th Annual Int. Symp. on Computer Architecture, pp.392-402, 1992.

[12] Keckler, S.W., Dally, W.J., "Processor coupling: integration of compile-time and run-time scheduling for parallelism"; Proc. 19-th Annual Int. Symp. on Computer Architecture, pp.202-213, 1992.

[13] King, P.J.B., "Computer and communication systems performance modelling"; Prentice–Hall 1990.

[14] Nemawarkar, S.S., Govindarajan, R., Gao, G.R., Agarwal, V.K., "Analysis of multithreaded multiprocessors with distributed shared memory"; Micronet Report, Department of Electrical Engineering, McGill University, Montreal, Canada H3A 2A7, 1993.

[15] Saavedra–Bareera, R.H., Culler, D.E., von Eicken, T., "Analysis of multithreaded architectures for parallel computing"; Proc. 2-nd Annual Symp. on Parallel Algorithms and Architectures, Crete, Greece, 1990.

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

[17] Tullsen, D.M., Eggers, S.J., Levy, H.M., "Simultaneous multithreading: maximizing on-chip parallelism"; Proc. 22-nd Annual Int. Symp. on Computer Architecture, pp.392-403, 1995.

[18] 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.