PERFORMANCE ANALYSIS OF ENHANCED FINE–GRAIN MULTITHREADED DISTRIBUTED–MEMORY SYSTEMS

W.M. Zuberek

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

Abstract

In fine–grain multithreading, the thread changes in each processor cycle, consecutive instructions are thus issued from different threads, and no data dependencies stall the pipeline. Enhanced fine–grain multithreading maintains a number of additional threads which are used to replace an active thread when it initiates a long–latency operation. Performance improvements due to enhanced multithreading are studied by analyzing a timed Petri net model of a fine–grain multithreaded architecture at the instruction execution level.

Keywords


1. Introduction

The performance of microprocessors has been steadily improving over the last two decades, doubling every 18 months (the so called Moore’s law [5]). At the same time, the performance of memory chips has been improving less than 10% per year [7], increasing the latency gap between the processor and its memory, and making it increasingly difficult to match the performances of the processor and the memory [10].

In distributed–memory systems, the latency of memory accesses is much more pronounced than in centralized-memory systems as memory access requests may need to be forwarded through several intermediate nodes before they reach their destination, and then the results need to be sent back to the original nodes. Each of the “hops” introduces some delay, typically assigned to the switches that control the traffic between the nodes [3, 4].

Instruction–level multithreading is a technique of tolerating long–latency memory accesses and synchronization delays in multiprocessor systems [1, 2], and in particular, in distributed–memory systems. The general idea is quite straightforward. When a long–latency memory operation occurs, the processor, instead of waiting for its completion (which in distributed–memory systems can easily require a hundred or more processor cycles), switches to another thread if such a thread is ready for execution. If different threads are associated with different sets of processor registers, switching from one thread to another can be done very efficiently [1, 3].

In fine–grain multithreading, the thread changes in every processor cycle [9]; this approach is advantageous for eliminating data dependencies that slow–down the processor’s pipeline; since consecutive instructions are issued from different threads, they have no data dependencies. Typically, the number of threads is equal to the number of pipeline stages, so no intra–instruction dependencies can stall the pipeline. In fine–grain multithreading, a thread issuing a long–latency memory operation becomes ‘waiting’ for the result of the requested operation. If a waiting thread is selected for execution, its slot simply remains empty (i.e., no instruction is issued), which is equivalent to a single–cycle pipeline stall. Since the threads issue their instructions one after another, fewer processor cycles are lost during a long–latency operation of a single thread.

In enhanced fine–grain multithreading, the number of available threads is greater than the number of pipeline stages, and the additional threads are used as a replacement of any active thread when it initiates a long–latency operation and becomes waiting. Consequently, the processor cycles are not lost, the utilization of processors increases and this improves the performance of the system.

A distributed memory system with 16 processors connected by a 2–dimensional torus–like network is used as a running example in this paper; an outline of such a system is shown in Fig.1.

It is assumed that all messages are routed along the shortest paths. It is also assumed that this routing is done in a nondeterministic way, i.e., if there are several shortest paths between two nodes, each of them is equally likely to be used. The average length of the shortest path between two nodes, or the average number of hops (from one node to another) that a message
must perform to reach its destination, is usually determined assuming that the memory accesses are uniformly distributed over the nodes of the system.

Although many specific details refer to this 16-processor system, most of them can easily be adjusted to other systems by changing the values of a few model parameters.

Each node in the network shown in Fig.1 is a fine-grain multithreaded processor which contains a processor, local memory, and two network interfaces, as shown in Fig.2. The outbound switch handles outgoing traffic, i.e., requests to remote memories originating at this node as well as results of remote accesses to the memory at this node; the inbound interface handles incoming traffic, i.e., results of remote requests that “return” to this node and remote requests to access memory at this node.

The execution of each issued instruction is modeled by transition $T_{run}$ (in the center of Fig.3). Consecutive instructions are issued by consecutive threads; this is modeled by the thread control section between $P_{ent}$ and $P_{next}$ and is discussed below. $P_{end}$ is a free-choice place with the choice probabilities reflecting the runlength, $\ell_t$, of threads (i.e., the average number of thread instructions executed between long-latency operations). In general, the free-choice probability assigned to $T_{net}$ is equal to $(\ell_t - 1)/\ell_t$, so if $\ell_t$ is equal to 10, the probability of choosing $T_{net}$ is 0.9; if $\ell_t$ is equal to 5, this probability is 0.8, and so on. The free-choice probability of $T_{end}$ is just $1/\ell_t$.

The selection of $T_{end}$ for firing indicates a long-latency memory access issued by the current thread. The access request (to local or remote memory) is placed in $Mem$, and a token is also deposited in $P_{wt}$ to indicate a possible thread replacement.

$Mem$ is a free-choice place, with a random choice of either accessing local memory ($T_{loc}$) or remote memory ($T_{rem}$); in the first case, the request is directed to $L_{mem}$ where it waits for availability of $Memory$, and after accessing the memory, the thread returns to the queue of waiting threads, $P_{rdl}$. Memory is a shared place with two conflicting transitions, $T_{rem}$ (for remote accesses) and $T_{lmem}$ (for local accesses); the resolution of this conflict (if both requests are waiting) is based on marking-dependent (relative) frequencies determined by the numbers of tokens in $L_{mem}$ and $R_{mem}$, respectively.

The free-choice probability of $T_{rem}$, $p_r$, is the probability of long-latency accesses to remote memory; the free-choice probability of $T_{loc}$ is $p_l = 1 - p_r$.

Requests for remote accesses are directed to $Rem$, and then, after a sequential delay (the outbound switch modeled by $S_{out}$ and $T_{sout}$), forwarded to $Out$,

2. Timed Petri Net Models

Petri nets have become a popular formalism for modeling systems that exhibit parallel and concurrent activities [8, 6]. In timed nets [12, 11], deterministic or stochastic (exponentially distributed) firing times are associated with transitions, and transition firings occur in real-time, i.e., tokens are removed from input places at the beginning of the firing period, and they are deposited to the output places at the end of this period.

A timed Petri net model of a fine-grain 4-threaded processor at the level of instruction execution is outlined in Fig.3, in which timed transitions are represented by solid bars and immediate transitions by thin bars.

The execution of each issued instruction is modeled by transition $T_{run}$ (in the center of Fig.3). Consecutive instructions are issued by consecutive threads; this is modeled by the thread control section between $P_{ent}$ and $P_{next}$ and is discussed below. $P_{end}$ is a free-choice place with the choice probabilities reflecting the runlength, $\ell_t$, of threads (i.e., the average number of thread instructions executed between long-latency operations). In general, the free-choice probability assigned to $T_{net}$ is equal to $(\ell_t - 1)/\ell_t$, so if $\ell_t$ is equal to 10, the probability of choosing $T_{net}$ is 0.9; if $\ell_t$ is equal to 5, this probability is 0.8, and so on. The free-choice probability of $T_{end}$ is just $1/\ell_t$.

The selection of $T_{end}$ for firing indicates a long-latency memory access issued by the current thread. The access request (to local or remote memory) is placed in $Mem$, and a token is also deposited in $P_{wt}$ to indicate a possible thread replacement.

$Mem$ is a free-choice place, with a random choice of either accessing local memory ($T_{loc}$) or remote memory ($T_{rem}$); in the first case, the request is directed to $L_{mem}$ where it waits for availability of $Memory$, and after accessing the memory, the thread returns to the queue of waiting threads, $P_{rdl}$. Memory is a shared place with two conflicting transitions, $T_{rem}$ (for remote accesses) and $T_{lmem}$ (for local accesses); the resolution of this conflict (if both requests are waiting) is based on marking-dependent (relative) frequencies determined by the numbers of tokens in $L_{mem}$ and $R_{mem}$, respectively.

The free-choice probability of $T_{rem}$, $p_r$, is the probability of long-latency accesses to remote memory; the free-choice probability of $T_{loc}$ is $p_l = 1 - p_r$.

Requests for remote accesses are directed to $Rem$, and then, after a sequential delay (the outbound switch modeled by $S_{out}$ and $T_{sout}$), forwarded to $Out$. 
Fig. 3. Instruction-level Petri net model of a fine-grain multithreaded processor.
where a random selection is made of one of the four (in this case) adjacent nodes (all nodes are selected with equal probabilities). Similarly, the incoming traffic is collected from all neighboring nodes in $Inp$, and, after a sequential delay (the inbound switch $Sinp$ and $Tsinp$), forwarded to $Dec$. $Dec$ is a free–choice place with three transitions sharing it: $Tret$, which represents the satisfied requests returning to their ‘home’ nodes; $Tgo$, which represents requests as well as responses forwarded to another node (another ‘hop’ in the interconnecting network); and $Tlocal$, which represents remote requests accessing the memory at the destination node; these remote requests are queued in $Rmem$ and served by $Trmem$ when the $Memory$ becomes available. The free–choice probabilities associated with $Tret$, $Tgo$ and $Tlocal$ characterize the interconnecting network [4].

The traffic outgoing from a node (place $Out$) is composed of requests and responses forwarded to another node (transition $Tgo$), responses to requests from other nodes (transition $Trmem$) and remote memory requests originating in this node (transition $Trmem$).

The thread control (upper left part of Fig.3) may look somewhat complicated, but it has a regular structure repeated for each represented thread. This basic structure, for thread “2”, is shown in Fig.4. The idea of this model is as follows. If the thread is active, a token is waiting in $Pth2$ for a ‘control token’ to appear in $Ps2$ (the marking of $Ps2$ in Fig.4 indicates that an instruction from thread “2” is going to be issued in the next processor cycle). Place $Ps2$ is an element of a ‘thread ring’ (in Fig.3 this ring connects $Ps1$, $Ps2$, $Ps3$, $Ps4$ and back to $Ps1$; there are several different ways connecting consecutive threads). This ‘thread ring’ contains a single token ($Ps1$ in Fig.3 and $Ps2$ in Fig.4).

If the selected thread is active, the firing of $Tth2$ inserts a token in $Pnth$ the next instruction to be executed by $Trun$, and another token in $Pr2$. If the issued instruction does not initiate long–latency operation, the free–choice transition $Tnth$ is fired (with the probability depending upon the thread runlength $t_t$), and a token is deposited in $Pent$. This token (together with a token in $Pr2$) enables $Twh$, firing of which regenerates a token in $Pth2$ and forwards the control token to $Ps3$.

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

If the queue of ready threads, $Prd$, is nonempty, transition $Tr2$ becomes enabled and its firing replaces the current thread by a new one, regenerating a token in $Pth2$ (immediate transitions take precedence in firing over the timed ones), so that another instruction will be issued when thread “2” is selected again. If, however, $Prd$ contains no tokens, the current thread remains ‘waiting’ for the completion of its long–latency operation (or for a ready thread entering $Prd$).

If a thread is ‘waiting’ (in $Pwu$) and a selection token appears in $Ps2$, the timed transition $Tw2$ fires and, after a unit of time (one processor cycle), deposits a control token in $Ps3$ (without issuing an instruction in this case).

The enhanced fine–grain multithreading is thus modeled by the initial marking of $Prd$, with the number of tokens representing the number of additional available threads.

3. Results

It is convenient to represent all timing information in relative rather than absolute units, and the processor cycle has been assumed as the unit of time. Consequently, all temporal data are expressed in processor cycles; e.g., $t_m = 10$ means that the memory cycle time $(t_m)$ is equal to 10 processor cycles, $t_s = 5$ means that the switch delay $(t_s)$ is equal to 5 processor cycles.

The utilization of 4–thread processors, as a function of $p_t$, the probability of long–latency access to local memory, is shown in Fig.5, where plot (1) corresponds to fine–grain multithreading without additional threads, plot (2) to enhanced multithreading with 2 additional threads, and plot (3) to enhanced multithreading with 4 additional threads. It can be observed that the effect of enhancements is more pronounced for values of $p_t$ close to 1 (i.e., when most of accesses are to local memory); for smaller values of $p_t$ (in this particular case) the interconnecting network determines the performance of the system, so the enhancements of the processor cannot have a significant effect. If 4
additional threads are available, the performance improvement is up to 30%.

Similar characteristics for an 8–thread system are shown in Fig.6. The effects of enhanced multithreading are less significant than for a 4–thread system.

The utilization of the input switch, in 4–thread and 8–thread systems, as a function of $p_{fr}$, is shown in Fig.7. The regions of low utilization of processors (in Fig.5 and Fig.6), i.e., the regions of small values of $p_{fr}$, corresponds to almost 100% utilization of the input switches, which indicates that the switches are the bottleneck of the system limiting its performance; the switches are simply too slow for this system.

Fig.8 shows the utilization of 4–thread processors, as in Fig.5 (so plot (1) corresponds to fine–grain multithreading without additional threads, plot (2) to enhanced multithreading with 2 additional threads, and plot (3) to enhanced multithreading with 4 additional threads), but in this case the switch delay is one half of that used in Fig.5. Consequently, the processor util-

![Graph](image1)

**Fig.5. Processor utilization for a 4–thread system**

($t_t = 10, t_m = 10, t_s = 10$).

![Graph](image2)

**Fig.6. Processor utilization for an 8–thread system**

($t_t = 10, t_m = 10, t_s = 10$).

![Graph](image3)

**Fig.7. Switch utilization; (1) 8–threads, (2) 4–threads**

($t_t = 10, t_m = 10, t_s = 10$).

![Graph](image4)

**Fig.8. Processor utilization for a 4–thread systems**

($t_t = 10, t_m = 10, t_s = 10$).

![Graph](image5)

**Fig.9.** The utilization of 8–thread processors as a function of $p_{fr}$, the probability of accesses to local memory, when the switch delay is one half of that used in Fig.5.

### 4. Concluding Remarks

The paper presents a timed Petri net model of fine-grain multithreaded multiprocessor system at the instruction execution level, and analyzes the effects of enhanced multithreading which maintains a number of additional threads to be used as replacements for threads which wait for the completion of their long-latency operations. It appears that a small number of such threads can quite significantly improve the performance of the system, but this improvement can be reduced by the presence of bottlenecks.

System bottlenecks can be identified by analysis of service demands for different components of the sys-
tem; the component with the maximim service demand is the bottleneck because it will reach the upper limit of its utilization first. For multithreaded multiprocessors, service demands can be considered with respect to a runlength of a single thread, i.e., to a sequence of instructions executed (by a single thread) between consecutive long–latency operations. In such a context, the processor’s service demand is equal to $l_\ell$ cycles, the service demand for memory (local and global, combined) is equal to $r_m$ cycles, the switch delay. For the case illustrated in Fig.7, the switch becomes the bottleneck when its service demand is approximately equal to 2 for a 16–processor system [4], the switch is the bottleneck for $p_v > 0.25$, or $p_v < 0.75$, which is well illustrated in Fig.7.

Although the discussion and presented results refer to a 16–processor system, the model needs only a few small changes to represent other multiprocessor systems. For example, the only changes that need to be made to represent a 25–processor or a 36–processor system, are the values of the free–choice probabilities associated with the transitions of $Dec$. A comparison of fine–grain multithreading with block multithreading is presented in [13].

Acknowledgement

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

References


Fig.9. Processor utilization for an 8–thread systems ($l_\ell = 10, r_m = 10, t_s = 5$).