

Performances of a Dynamic Threads Scheduler

Smaïl Niar¹ and Mahamed Adda²

¹ Université de Valenciennes BP 311 LAMIH-ROI
Valenciennes 59304 cedex France,
niar@univ-valenciennes.fr

² American University of Richmond, London
Queens' Rd. TW10 6JP, Surrey, England
addam@richmond.ac.uk

Abstract. This paper presents the design and development of a dynamic scheduler of parallel threads in the Multithreaded multiProcessor Architecture (MPA). The scheduler relies on an on-chip associative memory whose management time is overlapped with the execution of ready threads. The scheduler efficiently assigns resources to threads, and permits them to communicate with great flexibility. The results achieved with small number of threads from programs with high degree of parallelism are very satisfactory, even under various degrees of cache misses.

1 Introduction

The next generations of processors provide a very high degree of integration. The challenges focus around embedding several units of processing inside the chip and offering them enough useful work to benefit from these technological advances. However, because these processors are gaining in processing speed, memory units and interconnection network can no longer follow their progression. Thread level parallelism is one of the techniques adopted by computer designers to bridge the speed gap between processors in one side, and memories and communication networks in the other side. There exists so far two ways of integrating the multithread mechanism:

1. *Simultaneous multithreading* (SMT)[1][2], which consists in using a super-scalar platform containing several issue slots. These slots are filled by instructions selected from several active threads.
2. *Single chip Multiprocessor* (CMP) [3], where several processors of simple structure are integrated on the same chip. These processors co-operate on the execution of a program containing several threads.

The MPA (Multithreading multiProcessing Architecture) project that we present in this paper, although different in concept, follows the second [4]. With respect to other multithreaded architectures, the following points characterize our MPA:

- The processor utilizes a hybrid execution model: Control-flow to execute a thread, data-flow to synchronize communicating threads, and instruction-flow to pass messages between threads.
- The thread scheduling is performed automatically with parallel threads detected at compilation time. This scheduling does not impose any burden on the processors
- Each processor of the MPA network executes simultaneously several instructions from different threads. Only one execution pipeline is embedded in the processor.
- The communication between threads uses a message passing mechanism (via shared registers) that reduces the overhead when threads are not actives.

In this paper, we describe the mechanisms used by the MPA processors to extract and schedule parallel threads from parallel loops to obtain high level of performance.

2 MPA Processor Structure

In the MPA, to each thread we associate a number of resources called “*Context*” (Figure 1.(a)). Each context represents the state of a thread and consists of data registers, a PC and status registers [4].

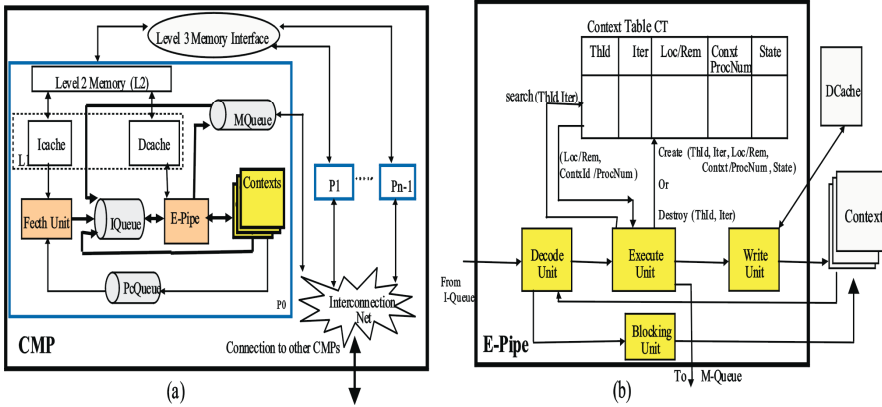


Fig. 1. (a) MPA processor internal structure in CMP environment and (b) its execution pipeline unit with the context table

The fetch unit continuously feeds the execution pipe (E-pipe) with ready instructions. The E-pipe consists of the decoding stage, followed either by the execution and the write stage or by the blocking stage fig 1.(b).

To maintain the synchronization between threads each data register is tagged with a presence bit **p** indicating the presence of an instruction or data. When an instruction, using a value sent by another thread, progresses through the E-pipe the availability of operands is checked before execution proceeds. If the operand is generated by another thread, it is written in the register. The arrival of an instruction requesting this data triggers the execution. When the operand is consumed, the register returns to the empty state. However, when the register is in an empty state, and an instruction requesting its data has entered the E-pipe, the **instruction** is

suspended and **written into the register**. The arrival of data activates the instruction by sending it to the Instruction Queue (IQueue) and data is copied into the register in one clock cycle. In general, instructions may arrive from the remote threads, from the Pc-Queue, or from the pool of previously suspended instructions. This technique supports migration of instructions, and hence gives a common view on the execution of threads, regardless of their locations.

The program in figure 2 shows an example to create 10 parallel threads of a loop.

```
//Adding 1 to all the elements of a vector X of 10 elements starting from address 100 in memory
Main :                               Add      #9, r0, r1                               //r1=9
NewTh:      Create      Increm, r1                               //Create a thread for executing
                                                // the Increm loop for iteration r1
                Add      r1, r0, r1, Increm, r1                    //send r1 to th. Increm[r1]
                Sub      #1, r1, r1                                // r1=r1-1
                Jump     pz, NewTh                                  //if (r1>=0) go to NewTh
                Destroy   main
/***** Loop *****/
Increm :
?          Load      #100, r1, r2                                //receive r1 from main and load
                                                // X[r1] in r2
                Add      #1, r2, r2                                // r2++
                Store    #100, r1, r2                              //store r2 in X[r1]
                Destroy  Increm
```

Fig. 2. An example of program written in the MPA language

3 Presentation of the Thread Scheduling

The scheduling of threads in the MPA processor is achieved by the usage of an on-chip associative memory called the context table (CT) Fig.1.(b). Except for the first thread in the first processor (the root thread), all the other threads are created and their entries stamped into the CT, when a thread executes the instruction CREATE. Each MPA processor maintains an active list of free contexts. When a thread is about to be created, the scheduler examines the list of local free contexts. If the list, is not empty the scheduler writes a new entry in the CT. However if the list is empty, the instruction CREATE is forwarded to an other processor of the CMP via the M-Queue, or is alternatively kept in the local processor until a context is freed. With the last approach we have examined several scenarios. When a CREATE instruction cannot be satisfied, the instruction is blocked in a special register or recycled in the I-Queue to be re-considered few cycles later. In the former, additional resources need be incorporated into the chip to hold instructions that cannot initiate a thread. In the latter, the instruction CREATE keeps circulating in the pipeline consuming useful cycles. The thread can carry on its execution even when there are insufficient resources (refereed to by CAF for “Continuation After Failure”). Another technique, consists of blocking the thread that initiated the unsatisfied CREATE. The instruction is written in the I-Queue and no subsequent instructions are fetched from this thread, until a context is freed (refereed to by SAF for “Stop After Failure”).

The access to the CT is only needed when an instruction is transferring a value to another thread, and this is reported in the instruction. The access is performed during the execution phase and it is overlapped with the computations involved in the ALU. The search key is represented by the couple (thread starting address, iteration number). If the destination context is not local, the CT supplies the address of the

remote processor. In the latter, the address of the destination processor and the result of the instruction is copied into the messages queue. The result is passed within a “migrating instruction”.

4 Experimental Results

In order to measure the performances of our MPA processor, we have designed an interactive graphical simulator. It enables the programmer to interactively change the parameters of the machine. In this paper, we expose the results attained from executing two programs on a single processor. Both programs are written in MPA language. In all the scenarios, we set the access time of the L1 caches to 1 clock cycle and the latencies of integer and floating functional units to 1 cycle.

4.1 Matrix Multiplication

Figure 3.(a) gives the execution time of the multiplication program of two matrices ($C = A * B$) function of the L2 memory latency in clock cycles. A, B and C are square matrices of 1600 (40*40) elements. The cache miss rate in L1 is fixed to 10% for all cases. Three versions of the matrix multiplication have been tested, corresponding to 1, 40, and 1600 threads extracted from the program. The first version (sequential) uses one thread and corresponds to the sequential program. In the second version (CAF40 and SAF40) the program calculates the matrix C in a line by line manner. As the processor has only 16 available contexts, a number of threads will be delayed until a context is freed. Before starting a new line of C, we ensure that all the previous threads have terminated their execution. Finally in the third version (CAF1600 and SAF1600), the program creates 1600 threads corresponding to 1600 points of the matrix C. We notice from figure 3.(a) that SAF method demonstrates higher performance over the CAF method. The later is very penalizing with large number of threads (1600). This is due to an increase in the number of recycling instructions.

In figure 3.(b), the latency of L2 is fixed to 10 cycles. Across all the spectrum of the cache misses, the method SAF remains the most significant over the method CAF and, in particular over the single threaded program (even with larger size L1 cache, miss = 0%). The single thread program pays a penalty of 2 clock cycles for every branch instruction. The SAF method on the other hand tends to fill these wasted cycles with some useful instructions from other threads.

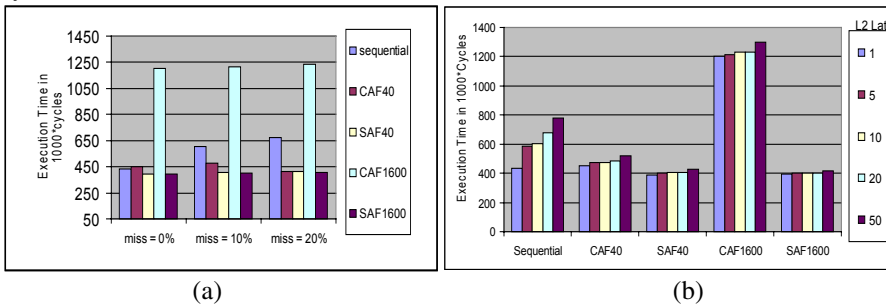


Fig. 3. Experimental results for the matrix multiplication program

4.2 DNA Sequence Alignment

In the second program, we implement the code of aligning two DNA sequences using Smith Waterman algorithm. In our multithreaded program, each thread calculates the elements of one column of the matrix result. In this section we use the SAF method. The size of the 2 sequences to align has been fixed to 100. Fig.4.(a) shows that for a L1 miss rate of 5% and 10%, the execution time increases barely with the memory latency. Even for higher cache miss, the increase in execution time is not dramatic, considering the increase of the memory latency. In Fig.4.(b) we have fixed the L1 miss rate to 10%. For all memory latencies, the execution time diminishes as the number of thread increases. When the number of contexts, goes beyond 16, the execution pipe operates at its maximum throughput. We can also notice that because of the thread management overheads in a single threaded program, the sequential version of the algorithm performs better.

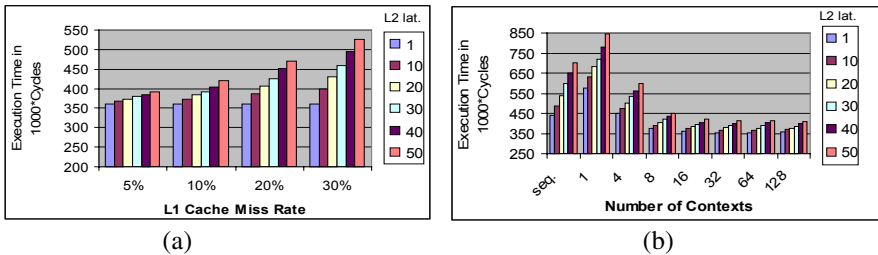


Fig. 4. Experimental results for the DNA sequence alignment program

5 Conclusions and Perspectives

In this paper we have presented the multithread scheduler for the MPA. The adopted approach is based on the utilization of an on-chip associative memory for stocking the information about the created threads. With a moderate number of parallel threads, it is possible to obtain higher performances in terms of execution time. The next stages of the project will be directed towards: the extension of the simulator to support multiple processors and to integrate speculative thread scheduling technique.

References

1. Tullsen, D. M. et al.: Simultaneous Multithreading: Maximizing on-chip parallelism. In Proc. Of the 22nd. annual. Intl. Symp. On computer architecture - 1995.
2. Marcuello, P., , González, A.: Exploiting Speculative Thread-Level Parallelism on a SMT Processor - Proc. of the Int. Conf. on High Perf. Computing and Networking - 1999.
3. Hammond, L., Nayfeh, B. A., Olukotun , K. : A Single-Chip Multiprocessor - IEEE Computer Special Issue on "Billion-Transistor Processors", p 79-85, September 1997.
4. Adda, M., Niar, S. : Thread Synchronization and Scheduling in a Pipelined Multithreaded Processor - 1st International Symposium on Advanced Distributed Systems - March 2000.