A Study of a Simultaneous Multithreaded Processor Implementation

Dominik Madoń[1], Eduardo Sánchez[1], Stefan Monnier[2]

[1] Logic Systems Laboratory
Swiss Federal Institute of Technology of Lausanne
INN-Ecublens 1015 Lausanne, Switzerland
Phone: +41 21 693 2676
Fax: +41 21 693 3705

[2] Computer Science Dept.
Yale University
51 Prospect Street
New-Heaven CT 06520, USA

Abstract. This paper describes an approach to the implementation and the operation of a Simultaneous Multithreaded processor. We propose an architecture which integrates a software mechanism to handle contexts, a rapid communication system, as well as a locking system to ensure mutual exclusion. We explain how the architecture manages the running threads as well as the software interface visible to the programmer. Finally, we provide a few indications on the efficiency of such an architecture.

1 Introduction

Over the past few years numerous publications have shown the advantages [3] [5][9][10] and sometimes the disadvantages [4][6] of Simultaneous Multithreaded (SMT) architectures. These architectures are based on a superscalar architecture to which several physical execution contexts are added. Such processors can thus simultaneously execute (and even issue) instructions originating from different processes.

The advantage of this type of processor lies in its better exploitation of functional units and above all in its ability to more efficiently handle the available memory bandwidth. While one process is busy fetching instructions from the main memory (prefetch), others can still execute instructions from the cache. or can launch their own prefetches, using a non-blocking memory bus. That's also where the improved use of memory bandwidth comes from. The latency between a fetch from the main memory or from a second-level cache can be hidden by the execution of instructions coming from other processes.

To date, a considerable amount of research has analyzed the advantages and disadvantages of this type of architecture. Yet, to the best of our knowledge, while several papers [8] have been published on possible instructions sets or on accessing registers of different contexts, nobody has studied the problems related to the design of a complete SMT processor, including all aspects of the implementation and in particular the creation of a process, its movement from memory to cache and vice versa, the destruction of a process, their synchronization, and the use of locking and other blocking mechanisms.

Also while the operation of SMT architectures in heterogeneous multitasking mode where multiple independent programs are executed has been studied quite extensively, little information is available on its operation in homogeneous multitasking mode [4]. In the latter mode, all the processes are created within a given application, requiring all the basic concurrent programming commands (for example, means to communicate and to synchronize efficiently and cheaply, in addition to a locking system for the protection of sensitive data).

This paper would like to shed some light on the above issues by proposing an original approach to the use of SMT processors. We start by enumerating the functions that need to be made available to the programmer to enable her to use the processor and handle the physical contexts which host the logical processes. In order to implement these functions, we then describe an architecture which includes a management system which uses ports. Next, we will discuss the implementation of the communication between contexts by presenting a solution for the rapid transmission of data which we will then extend to the creation of locks. Then we will explain how to physically implement these two features, starting with a description of a functional simulator which we use to validate our hypothesis and obtain some preliminary results. In this section we also discuss the physical limitations of the architecture which, up to a certain point, determine the possible choices for the implementation. This article ends with a conclusion on the future direction of our research.

2 Basic definitions used in parallel applications

2.1 Contexts

An *execution context* is the group of registers used by a task during its execution: floating-point registers, fixed-point registers, the program counter, the various state registers, and the different configuration registers (for example the register which stores the configuration of the Memory Management Unit), for access to memory). A *physical context* corresponds to the registers provided by the hardware to store an execution context in the processor.

In a single-task processor there is only one physical context, but it can have several execution contexts, one per process. In a such processor, at any given moment, one process is executed. In a SMT processor, on the other hand, several physical contexts allow the execution of several processes simultaneously.

2.2 Context switching

In most cases, an initial process is charged with the "placement" of execution contexts within the physical contexts available. This initial process assigns other processes to the available physical contexts, independently of the type (homogeneous, heterogenous or a combination of both) of multitasking. The processes launched by the initial process can in turn assign processes to unused physical contexts, and so on. To launch new processes, processes thus need to access physical contexts other than their own.

On the processor, the processes are multiplexed on physical contexts. Therefore, a "master" process can demand that a "slave" process be suspended and its physical context be saved to memory. To do this, the master process must thus be able to suspend and resume the execution of a physical context. The suspension implies that the pipeline of the processor must halt the execution of the instructions belonging to the suspended physical context. The resumption implies that the pipeline of the processor is again authorized to execute the context's instructions.

2.3 Synchronous data exchange

The execution of a set of processes originating from a single application may include the exchange of data, the synchronization between the different processes, and the mutual access which is only available for certain shared data. The exchange of data can occur through shared main memory, a solution which is very practical when large quantities of data are being transferred but can be time-consuming. Indeed, the data has to transit through, in the best case, a shared data cache, or, in the worst case, a distributed data cache which may raise cache consistency issues (which in turn may cause the suspension of the receiving process) and require a snooping mechanism on the caches. This data exchange method proves to be very time-consuming (latency) if the amount of data to be transfered is small (one or several words) and the transfer is synchronous, in which case much more efficient methods are available. Indeed, more instruction cycles will be used to synchronize the processes than to transfer the data between them. This will be more time-consuming especially if one of the processes is not ready implying thus that the second one has to be suspended until the former is ready.

2.4 Locks

Locks are used to control the access to protected resources. The code that accesses the resource is delimited by lock and unlock instructions. In many processors, locks are built using atomic instructions or by masking interruptions. If a lock is already in use, the process wishing to access the resource is suspended until that lock is unlocked. As in the paragraph above, the process to be resumed may no longer be in the physical execution contexts.

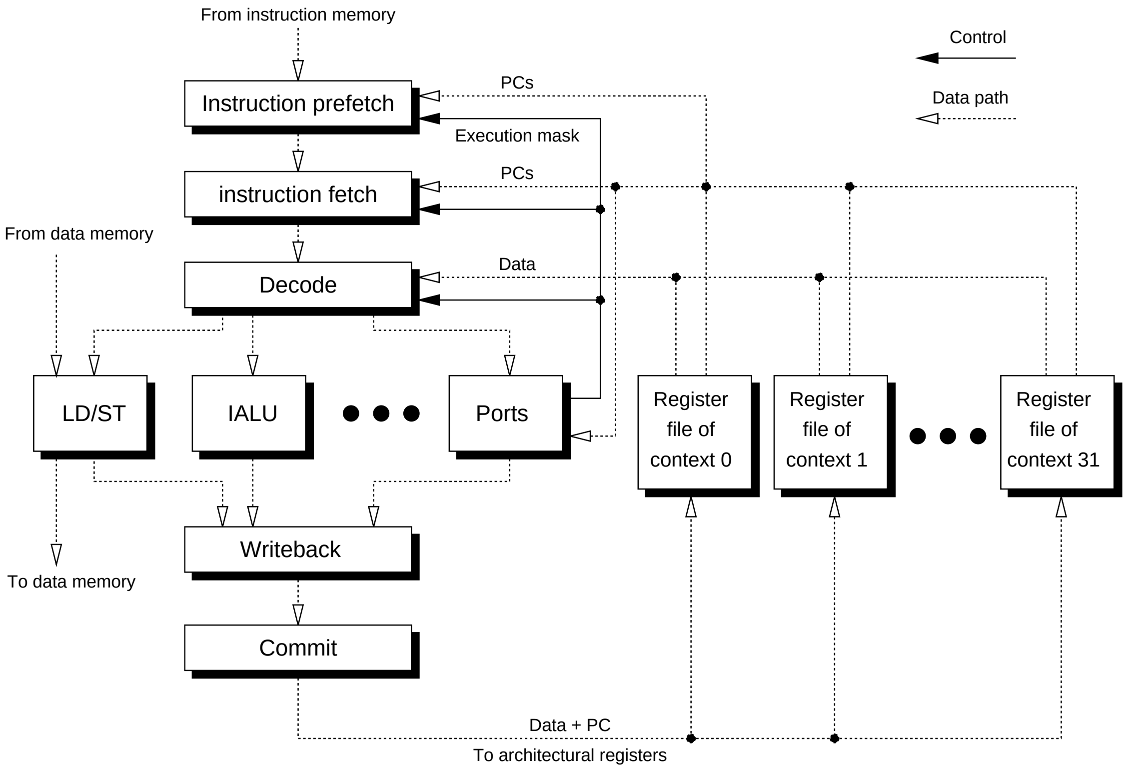


Fig. 1. Our *Simultaneous Multithreaded* Architecture

3 Processor architecture

3.1 Superscalar foundation

For this study, we chose as a starting point a well-known 5-stage (fetch, decode/dispatch, execute, writeback, commit) superscalar architecture [7] (figure 1). The instructions are executed out-of-order, using speculative branching and loading, with a one-level prediction table for the branches. The processor contains the following execution units:

- 4 integer ALU units
- 1 integer multiplication/division unit
- 2 load/store units
- 4 floating-point ALU units
- 1 floating-point multiplication/division unit

The processor holds 32 32-bit integer registers, 32 32-bit floating-point registers (usable by pairs as 64-bit registers), a floating-point state register, and a PC.

The processor holds two first level cache memories, one for instructions and the other for data, and a unified second level cache memory.

3.2 Proposed extensions

The first modification to migrate to a SMT architecture was to duplicate 32 times the physical context of the original processor, thus allowing 32 execution contexts to execute simultaneously.

We then added a *prefetch* stage to the pipeline to determine which tasks' instructions will be loaded from the memory (the first level cache) into the queue of instructions to be decoded the next clock cycle(fetch stage). This stage, using a mechanism that detects the presence of the instructions the in cache, can foresee cache misses and thus give precedence to tasks which will not block the pipeline. This is the stage that best exploits the data independence between contexts executing different program segments [3].

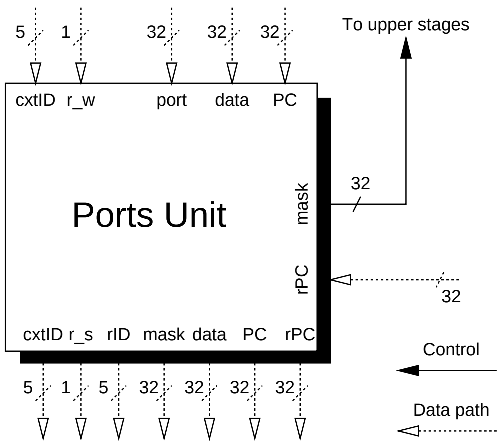


Fig. 2. Functional unit managing contexts/ports/locks

We also added a functional unit (see figure 2) in the execution stage which allows for three types of instruction.

This functional unit takes care of informing the various stages of the processor of the activation stage of the contexts execution mask. The instructions belonging to active contexts can go through the pipeline to be executed. The suspended contexts do not have access to the pipeline. The instructions already executed (i.e., in the writeback or commit stages) before the arrival of blocking instruction have the right to complete, while the others are flushed out of the pipeline, exactly as for a branch prediction error.

- Instructions which affect physical contexts (e.g. to stop, start, read, or write the PC of a given context, or to access the port on which a context is waiting);
- Instructions to exchange data between and synchronize two physical contexts;
- Instructions to exploit physical locks.

3.3 Operation of the extensions

The placement of these three functions in the same unit has been motivated by the similarity of their actions on the ordering of the instructions (see figure 3). Locks and synchronous data exchanges both require the ability to start and stop the execution of physical contexts (i.e. access to the context's PC). This control should be centralized as it implies the suppression of instructions in the decode/dispatch, writeback, and commit stages (invalid instructions).

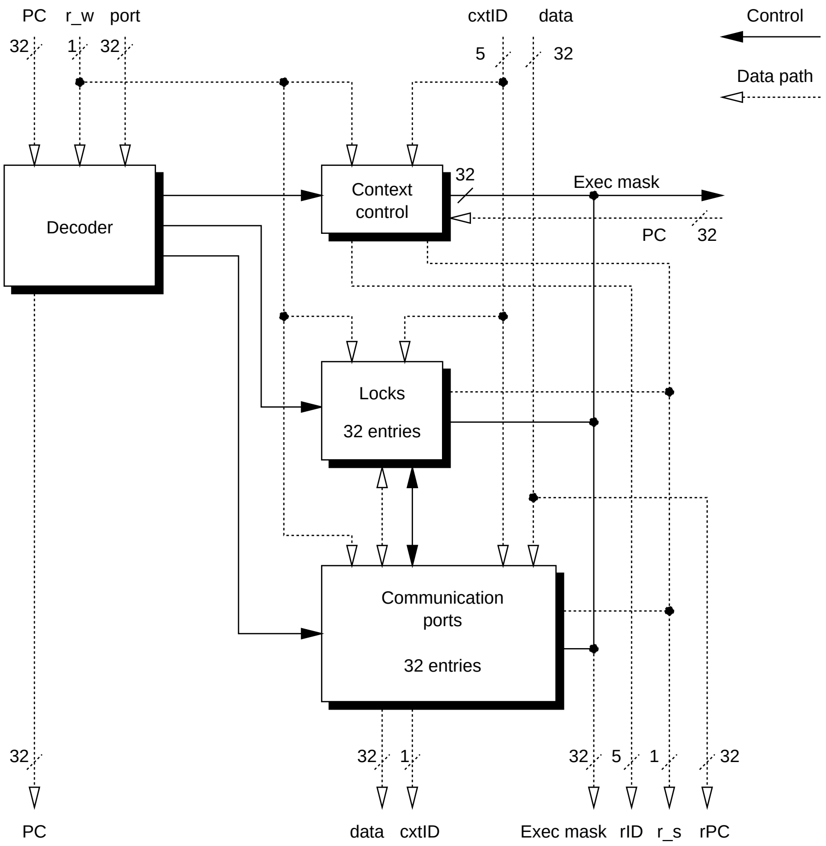


Fig. 3. Functional unit managing contexts/ports/locks

This unit should be used in the same fashion as the load/store unit when it has to write a word back into the main memory: the instructions sent to this unit have to be executed solely in non-speculative mode to avoid inconsistent results. For instance, if a context sends a message immediately after the execution of a branch, the message can be delivered only if that branch is taken. In fact, the simultaneous execution of instructions coming from different contexts does not guarantee that the message will not be read between the branch and the speculative delivery of the message. If the branch was wrongly predicted, it would then be necessary to correct the context which read the message and invalidate all the instructions since the reading of the message.

3.4 Instruction set extensions

To access the context control and communication/lock unit, we added two instructions to the instruction set allowing a context to read from and write to a port. These two instructions use two registers, one for the data and one for the port number:

```
rport   Rd, Rp    # read from port Rp into Rd
wport   Rs, Rp    # write Rs to the port Rp
```

The register `Rp` holds the number of the port to access. The register `Rd` holds the value read from the port. The Register `Rs` holds the value to send to the port. Any of the 32 general purpose registers may be used for `Rp`, `Rd` et `Rs`.

The port numbers are coded on 32 bits. However, some codes are reserved for special use: data exchanges through ports, for instance, require numbers greater than 256 (`0xff`). The list in table 1 provides some information on the numbers reserved for special functions belonging to three separate categories described in the following three sections.

These operations must be idempotent to avoid problems in context switching. We will see later how this characteristic is implemented and how to best exploit it.

It is also useful to note that the three categories of functions could be identified through the coding of the instruction. This solution is of course preferable provided that the instruction set allows it. For the user, this would simplify the access to the communication ports and to the locks, the numbers no longer identifying the function but only the particular port or lock.

3.5 Execution control of another context

The execution or the suspension of a process may be controlled by the process itself or by a different process. When the request is external, it is very often the case that it was motivated by a context switch. Similarly when a context is started by another, it is very likely that a context switch just happened (unless it is as a result of an exchange of messages between two processes in two different physical contexts or of the release of a lock).

The context switches, that is to say the transfer of registers from a physical context to the memory or the contrary, may be controlled by the context that is leaving. The access (both writing and reading) to another context's PC and the existence of a shared memory area are sufficient to perform this task. More precisely, the two operations needed are:

– Reading a context's PC gives back its value and suspends it on the instruction pointed by the PC. The context will then resume from that same instruction address. address.
– Writing into a context's PC causes it to resume.

Starting a new process in a context (figure 4) is done by preparing the associated stack and inserting in it the execution context. After that, the address of a minimal loading procedure is written into the PC of the context which is going to execute the process. That process starts and waits on a specified message port. The initiating process then writes to this particular port the stack pointer. The starting context reads this value, loads the execution context into the registers and jumps to the portion of code it should execute.

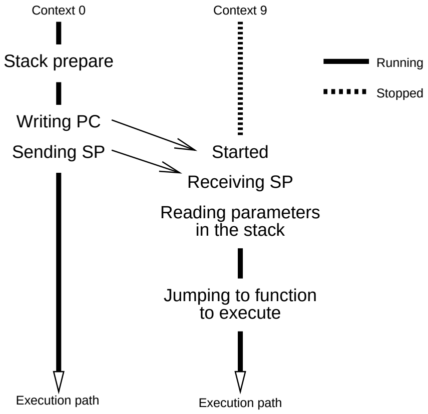


Fig. 4. Process starting

A context switch occurs in the same way. A process (which we will call *master*) which decides to suspend another process (the *slave*) writes into the PC of the *slave*'s context the address of the procedure which moves the context back to the memory. This causes the *slave*'s registers to be written to the stack and its stack pointer to be sent as a message back to the *master*. At this point the physical context is ready for use by another process. It is then sufficient to write into the PC the address of the loading subroutine for the incoming process. This subroutine first reads a message on the port where the *master* process writes the value of the appropriate stack pointer. With it, it is possible to retrieve the correct execution context. Once the value of all the registers has been restored, the *master* can write into the PC the address of the instructions executed by the resuming process at the time it was suspended.

The messages, as we have just seen, are short and separated by a variable number of instructions which may go from one to several dozens.

Thus, to stop, start, or switch a context, the ability to access to another context's PC and the existence of a data transmission mechanism are sufficient.

3.6 Synchronous communication by port

The transmission of data between contexts can be done, as we mentioned, by sharing a portion of memory. This mechanism, however, is not efficient when transferring only a few words: nothing guarantees that the information sent will remain in the cache until it is requested by the destination context.

A solution consists of using distributed caches, which require a snoopy mechanism to detect when a context is trying to access data which is stored in the cache of another context (and whose value is not necessarily the same same as in the main memory). The use of the cache in writethrough mode solves this cache coherency problem but increases the transfer time of the data (which will have to go from the processor to the memory and back again).

The exchange mechanism that we propose relies on the notion of communication ports. A communication port is a one-word-wide data register, associated to a port number. Reading or writing to a port is blocking (until execution of the corresponding write resp. read) allowing synchronous exchanges of data.

The context management unit thus contains an associative table of 32 entries (it is useless to have more entries than contexts) that allows the implementation of these communication ports. The format of the table's entries is described in figure 5.

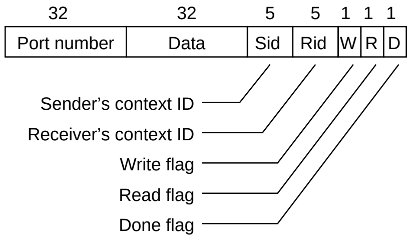


Fig. 5. Entry in the table of communication ports

Each line contains a field for the number of the port used, a field for the value, two fields for the physical numbers of the contexts which are using this entry (the sender and the recipient), and three status flags.

This entry format allows several contexts to use the same port number, in which case the requests will be satisfied in an arbitrary order.

Sending and receiving are almost symmetrical from the point of view of their operation. What follows is a description of what occurs when context 0 writes in port 3055 and context 9 later reads from the same port (figure 6).

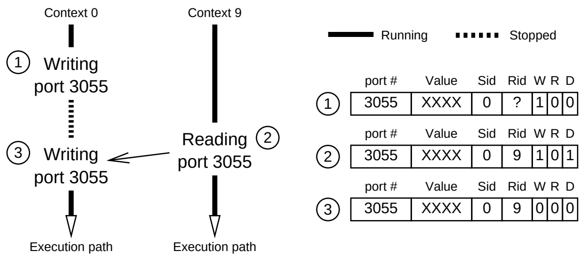


Fig. 6. Communication by synchronous port

1. The first instruction executed by context `0` writes to an available entry (all flags are `0`) the port value `3055`, the datum `XXXX` to be transferred, and its physical number (`0`) in the `Sid` field. It then sets the `W` flag to `1`, indicating that a context is waiting for this port to be read. The execution of the sender (context `0`) is then suspended.
2. When the recipient is ready to read the datum, it writes its number in the appropriate field (`Rid`), sets the `D` flag (done) to `1`, reads the value, and prompts the sender to resume its execution. Flag `D` prevents all other processes from reading this datum and will tell the sender that the value has been read and that it may clear the entry.
3. Following its resumption, the sender executes again the instruction, which, seeing that the write flag `F` and the done Flag `D` of port `3055` are both `1` and that the `Sid` field contains its own physical number, will clear all flags and thus free the port.

It will be noted that this structure allows any context to be suspended if it is waiting on a port. The process, when it resumes, will execute the same instruction (send or receive), which will again result in either the exchange of data or another suspension. It will be noted that a context which owns an entry where two flags (`W` and `D` or, `R` and `D`) are `1` will not be stopped or will not be able to change PCs until the re-execution of the communication instruction which erases all the flags. This mechanism ensures that a process cannot stop a context that has completed a message exchange but has not yet re-executed the `rport` or `wport` instruction in order to increment its PC.

3.7 Hardware locks

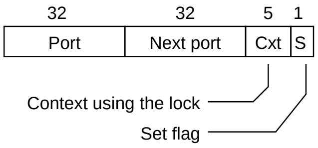


Fig. 7. Locks table's entry

Synchronous ports are meant for message exchange but can of course be used to implement locks. This solution is inconvenient in that it requires auxiliary contexts (since a communication port links two contexts, a sender and a recipient, while locks concern only a single process). Moreover, locks should also operate as fast as possible. For these two reasons we have created a *lock table*, whose entries have the format described in figure 7. These locks allow critical sections to be protected using the access instructions `wport` and `rport` of the context control unit. The suspension of a context when a lock prevents access, uses communication ports as a chained list (queue) of waiting contexts. This method allows us both to reuse a structure already available and to transfer to memory contexts waiting on a lock.

Each entry from the lock table (32 in all) contains a communication port number for next context wishing to access the critical section, a communication port number for the last context on standby in the queue, the number of the context which has set the lock, and a flag `S` (set) indicating if the lock is available or busy.

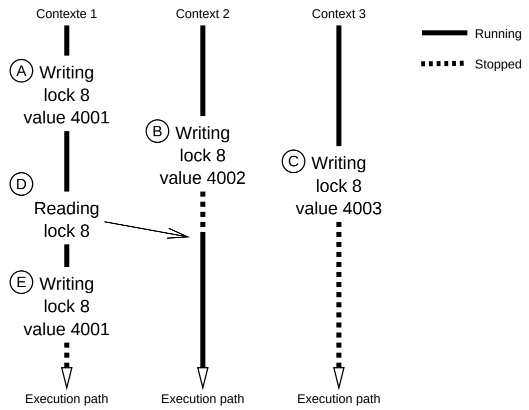
To explain the operation of the lock, let us imagine the following situation: three contexts (`1`, `2`, and `3`) ask in turn to set lock `8` (see table 1 for the addressing of the locks). Context `1` then releases the lock, and immediately request to again(figure 8):

A. Context `1` sets the lock, formerly available, with the following instructions:

```
wport 4001, 104       # grabbing port 8 (104 - 96)
# Start of the critical section
                 :
# End of the critical section
rport   4001, 104
```

This instruction sets lock number `8` (corresponding to port `104`) by setting the `Set` flag to `1`, writing its context number (`1`) in the `Cxt` field, and setting the `next port` field to `4001`, indicating that the next context wishing to access the locked section should use port `4001`. It then proceeds and starts executing the critical section.

B. The second context wishing to access the critical section (context `2`) will try to write to port `104`, find the `Set` flag is `1`, indicating that the critical section is not immediately accessible. It will then read the next port number (`4001`) to find where the queue ends. It will create a new port entry, using the port number it has retrieved (`4001`), its own context number (`2`) for the `Sid` field, and setting the write flag to `1`. To indicate that the end of the queue has changed It will also write the new port number (`4002`) both in the next port field of the lock and in the value field of the port. Context `2` will then be suspended until port `4001` is released.



		port #	next port #	Cxt	S
A	8	4001	4001	1	1
B	8	4001	4002	1	1
C	8	4001	4003	1	1
D	8	4002	4003	2	1
E	8	4002	4001	2	1

	port #	Value	Sid	Rid	W	R	D
B	4001	4002	2	?	1	0	0
C	4001	4002	2	?	1	0	0
	4002	4003	3	?	1	0	0
D	4002	4003	3	?	1	0	0
E	4002	4003	3	?	1	0	0
	4003	4001	1	?	1	0	0

Fig. 8. Operation of locks: an example

C. When the third context asks to access the critical section, step 2 above is repeated (a port entry for port number `4002` is created and the end of the queue moves to `4003`).

D. To exit the critical section, context `1` will issue the following instruction: `rport 4001,104`. This instruction will access port `104`, find that the port (`4001`) and the next port field (`4003`) are different (i.e., that at least one other context is waiting to access the critical section). It will then access the port entry corresponding to port number `4001`, where it will find the number of the context on standby (`2`) and the port number for the next context in the queue (`4002`). it will modify the lock entry using these two numbers, and then release the port by clearing the write flag, which will cause context `2` to resume execution. Upon resumption, context `2` will access port `104` and find that it has control of the critical section.

E. Context `1`'s request to enter anew into the critical section will be handled like any other request, and the modifications to the two tables will occur as above.

Port number	Access type	Function
`0x00000000`	Read	Suspend the caller
`0x00000001`	Read	Return the execution mask, each bit represents a context. The bit position in the word gives the context number. If the bit is set the context is running else the context is suspended
`0x00000002-0x0000003f`		Unused
`0x00000060-0x0000007f`	Read	Release a lock
`0x00000060-0x0000007f`	Write	Set a lock
`0x00000080-0x0000009f`	Read	Read the PC of the context number *port - 0x80* and suspend it
`0x00000080-0x0000009f`	Write	Write the PC of the context number *port - 0x80* and resume it
`0x000000a0-0x000000bf`	Read	Returns the port currently used by context number *port - 0xa0*. If the context is not used by a port 0 is returned. If the context was using a context on a port, this port is released
`0x000000c0-0x000000df`	Read	Return the MMU configuration for context number *port - 0xc0*
`0x000000c0-0x000000df`	Write	Write the MMU configuration for context number *port - 0xc0*
`0x000000e0-0x000000ff`		Unused
`0x00000100-0xffffffff`	Read Write	Communication ports

Table 1. Port description

4 Simulator description

To rapidly obtain experimental results without physically implementing a processor and its corresponding software, we have decided to use a set of tools developed for the construction of processor architectures: the SimpleScalar tool set [1][2]. This package contains many C libraries allowing the construction of a functional processor simulator. It also contains a C compiler (gcc) allowing the generation of machine code with a predefined (but extensible and modifiable) set of instructions.

The architecture described above has been implemented using SimpleScalar. The simulator, called SSMT (SimpleScalar Multithreaded), has been tested with the help of a few benchmarks. We have thus verified that results obtained were similar, in terms of instructions per cycle (IPC) performance, to those of other teams [10].

Although the simulator allows an infinite number of different configurations, certain architectural parameters remained unchanged. We concentrated particularly on adapting the simulator to our extensions and on their test. For the latter we have developed several routines in assembler to exploit the processor, as well as the corresponding C interface.

The results which interest us are derived, for the most part, from the traces obtained from the execution of programs. The analysis of these traces allows us to measure the performance of different configurations and average them. In fact, since the SMT architecture allows the execution of multiple processes, the execution of a process in a context changes depending on the number of contexts already running and on what they are executing.

5 Preliminary evaluation

The architecture we have described was first tested to determine its efficiency in the execution of independent processes (heterogeneous multitasking), which has allowed us to verify the important results obtained by Tullsen et al. We then began to evaluate the homogeneous multitasking performance.

The central idea of this architecture is based on the execution of several concurrent processes. Performance superior to that of superscalar processors cannot be reached without a sufficiently important degree of parallelism. Nevertheless, we know that the use of parallelism involves a certain increase in the number of instructions to be executed (creation and destruction of processes, message passing, etc). The efficiency, therefore, depends also on the tools at the programmer's disposal. If the creation of a process is not too time-consuming (compared to the calling of a function, for example), function calls might be replaced by parallel execution, where possible.

To compare their performance we have generated several traces on programs executing either function calls or spawning processes. It is difficult to produce accurate numbers since the execution depends on the surrounding instructions, and on other tasks in the case of multitasking. Here we have measured the number of cycles necessary for a function call (from the jump instruction to the first instruction of the function) for a context executing by itself. For the creation of a process, we used two active contexts: the first creates the second which executes the function. In both cases the instructions were in the cache.

Table 2 shows the ratio of the number of cycles between the two methods (process creation/function call). As an example, the creation of a process required a minimum of 17 cycles.

We have also measured the performance with a greater number of contexts. In this case, with explicit context priorities not implemented, the results show a greater disparity in the number of cycles necessary. A simple function call may require 24 cycles while the creation of a context require 111. In this configuration, the worst ratio is 4.63. Evidently this type of measurements do not allow precise conclusions to be drawn regarding the performance of the creation of processes.

The measurements of the speed of context switches in our architecture, compared with a simply superscalar version, also exhibits this type of variance which depends exclusively on the execution of other contexts.

Nevertheless, if the synchronization by port, the message passing, and the use of hardware locks require few execution cycles (the ports management unit executes an instruction in at most two clock cycles), the measurements taken at this level are afflicted by the same type of problems.

For all these reasons we are orienting ourselves to other types of measures based on multi task kernel written in C. These will allow us to obtain higher-level information such as, for example, the efficiency of our processor for a classical paradigm such as the producer-consumer problem.

measure	Ratio
minimum	2.11
maximum	2.43
average	2.25

Table 2. Ratio between a function call and the start of a process

6 Conclusions

We have studied most of the issues related to the implementation of an SMT processor. Our solutions have been validated on a simulator and the first results are very encouraging, pushing us to complete a physical implementation.

The evaluation of the performance of our solution remains incomplete, as we have mentioned. We are currently comparing our architecture with single-processor and SMP (Symmetric Multiprocessor) solutions as well as with other types of multitasking architectures. This comparison is based on high-level concepts (related to the paradigms of concurrent programming).

Nevertheless, the scarcity of SMP or multitasking systems makes it difficult to obtain a good evaluation of the performances of SMT hardware as opposed to those of classical superscalar solutions, particularly since a good compiler generating parallel code remains elusive. While other contributions have shown that the effective output of an SMT processor is greater than that of a standard superscalar processor, it cannot be stated with certainty that the result will be as good in homogeneous multitasking mode with code generated by a parallelizing-compiler. In this case, in fact, the quality of the produced code (the decomposition in different processes) and the efficiency of process creation/destruction tools play an important role.

The SMT simulator and the software tools we have developed allow us, at the present time, to continue our work in two parallel directions, the first being the completion of the processor through the addition of interruptions, and the second a more exhaustive evaluation of the processor.

Acknowledgments

We would like to thank the many peoples who have contributed to this project, and more particularly Professor Eduard Auguadé, of the Polytechnic University of Catalonia (UPC) at Barcelona, for his advice.

References

1. Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-96-1308, University of Wisconsin, Madison, July 1996.
2. Douglas C. Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.
3. S. J. Eggers, J. Emer, H. M. Levy, J. L. Lo, R. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. Technical Report TR-97-04-02, University of Washington, Department of Computer Science and Engineering, April 1997.
4. A. Farcy and O. Temam. Improving single-process performance with multithreaded processors. In *Proceedings of the 1996 International Conference on Computing*, pages 350–357, New York, May25–28 1996. ACM.
5. Manu Gulati and Nader Bagherzadeh. Performance study of a multithreaded superscalar microprocessor. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, pages 291–301, San Jose, California, February 3–7, 1996. IEEE Computer Society TCCA.
6. Sébastien Hily and André Seznec. Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading. Technical Report PI-1086, IRISA, University of Rennes 1, 35042 Rennes, France, February 1997.
7. Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.
8. Mat Loikkanen and Nader Bagherzadeh. A fine-grain multithreading superscalar architecture. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 163–168, Boston, Massachusetts, October 20–23, 1996. IEEE Computer Society Press.
9. Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice : Instruction fetch and issue on an implementable simultaneous MultiThreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecure*, pages 191–202, New York, May22–24 1996. ACM Press.
10. Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA.