

Relatório Técnico

**Núcleo de
Computação Eletrônica**

NCESPARC+: An Implementation of a SPARC Architecture with Hardware Support to Multithreading for the Multiplus Multiprocessor

J. S. Aude
F. R. S. Martins
M. A. S. Barbosa
M. João Jr.
M. T. Young
S. B. Pinto

NCE - 27/99

Universidade Federal do Rio de Janeiro

NCESPARC+: An Implementation of a SPARC Architecture with Hardware Support to Multithreading for the Multiplus Multiprocessor

J. S. Aude¹, F. R.S. Martins¹, M. A. S. Barbosa²,
M. Joao Jr.², M. T. Young², S. B. Pinto²

¹Federal University of Rio de Janeiro, NCE and IM

²Federal University of Rio de Janeiro, NCE

P.O. Box 2324 - Rio de Janeiro - RJ 20001-970 - Brasil
{e-mail: salek@nce.ufrj.br}

Abstract

NCESPARC+ is an implementation of the SPARC V.8 architecture with hardware support to a variable number of thread contexts, which is under development for use within the framework of the Multiplus distributed shared-memory multiprocessor. It is expected to provide an efficient and automatic mechanism to hide the latency of busy-waiting synchronization loops, cache-coherence protocol and remote memory access operations within the Multiplus multiprocessor. NCESPARC+ performs context-switching in at most four processor cycles whenever there is an instruction cache miss, a data dependency in relation to the destination operand of a pending load instruction or a busy-waiting synchronization loop. It has a decoupled architecture which allows the main pipeline to process instructions from a given context while the Memory Interface Unit performs memory access operations related to that same context or to any other context. Results of simulation experiments show the impact of some architectural parameters on the NCESPARC+ processor performance and demonstrate that the use of multiple thread contexts can effectively produce a much better utilization of the processor when long latency operations are performed. In addition, NCESPARC+ processor performance with a single context is superior to that of a standard implementation of the SPARC architecture due to its decoupled architecture.

Keywords: Multithreaded Architectures, SPARC Architecture, Context-Switching, Latency Hiding

1. INTRODUCTION

NCESPARC+ is a specially designed multithreaded SPARC processor to be used within the Processing Elements of the Multiplus distributed shared memory multiprocessor [Aude96]. The motivation for the development of NCESPARC+ is to conceive an efficient solution to hide the latency of remote memory access operations in large scale distributed shared memory multiprocessors. Within the Multiplus architecture, a long latency memory access operation may typically extend for over 100 processor cycles, forcing the processor to sit idle waiting for the operation completion. Less time would be wasted if the latency could be partially hidden by the execution of some useful task set to run through a fast enough context-switching operation.

Multithreaded processor architectures can greatly reduce the context-switching overhead by providing multiple hardware contexts, that is, multiple sets of General Purpose Registers, Program Counters (PC's) and Processor Status Registers (PSR's). They are designed to support context-switching either on every processor cycle (fine-grain multithreading) or on an event which

may cause latency (coarse-grain multithreading), such as cache misses, load/store instructions, etc.

The Sparcle processor designed at MIT for the Alewife multiprocessor system [Agar99] is a multithreaded architecture based on the SPARC processor in which a single PC and a single PSR are used for all the four available hardware contexts. Context-switching takes place on every cache miss (coarse-grain multithreading). Since, the PC and the PSR contents have to be saved in memory, the context switching overhead is still 14 processor cycles long.

On the other hand, the MTA architecture [Byrd95] designed by Tera Computer Corporation supports 128 hardware contexts with separate register files and can perform context switching between threads on every processor cycle (fine-grain multithreading). At any time instant, a single instruction of a particular thread context is present in the pipeline. With this approach, the pipeline design is greatly simplified, since data dependencies do not occur, but the performance of sequential codes is heavily penalized. Laudon, Gupta and Horowitz [Laud94] have proposed the Interleaving approach to allow fast execution of a single thread or multiple threads under the fine-grain multithreading scheme. With this technique, different instructions belonging to the same thread context can be simultaneously present in different stages of the pipeline. When context switching is performed, only the instructions related to the context to be switched out of the pipeline need to be squashed. This technique adds some hardware cost to the implementation of a standard fine-grained multithreaded processor, but it is specially effective for use with deep pipelines within a multiprogrammed uniprocessor.

NCESPARC+ is an implementation of a coarse-grain multithreaded SPARC V8 architecture [Weav92] which can support a variable number of hardware contexts, from 1 to 16, with a worst case four processor cycles context-switching overhead. It is targeted for use within large scale distributed shared memory multiprocessors. Within NCESPARC+, a thread keeps issuing instructions to the pipeline until a first level instruction cache miss occurs, data dependency in relation to a previous pending load operation is detected or a busy-waiting synchronization loop occurs. At this point, the context is switched after a single processor cycle to another thread which is ready to run. At most, three instructions already present in the processor pipeline will have to be annulled.

Section II of this paper briefly reviews the Multiplus distributed shared memory architecture and gives information on its typical latencies for remote memory access operations. In Section III, the basic features of the NCESPARC+ architecture are presented. The main components of the NCESPARC+ architecture are described in Section IV. In Section V, the different context-switching operations that can take place within NCESPARC+ are discussed in detail. Section VI presents and discusses some simulation results considering an application which performs the inner product of two integer vectors. These results indicate the impact of some architectural parameters on the NCESPARC+ processor performance and show that, with a proper configuration, NCESPARC+ can effectively hide the latency of long remote memory access operations. In addition, the simulation results show that even when a single context is used, NCESPARC+ performs better than the standard SPARC architecture. Finally, in Section VII some conclusions and directions for future work are presented.

2. THE MULTIPLUS ARCHITECTURE

Multiplus is a distributed shared-memory multiprocessor based on the interconnection of clusters of Processing Elements (PE's) as shown in Figure 1. The Multiplus Processing Element is based on the use of SPARC processors. The current implementation of the Processing Element uses the Cypress SPARC chipset and supports separate 64-Kbyte instruction and data level 1 caches and up to 32 Mbytes of memory belonging to the global address space. Within Multiplus, a set of Processing Elements and one I/O Processor can be interconnected through a 64-bit double-bus system making up a cluster. The first bus is dedicated to instruction and data access operations and the other one is only used to perform block transfer operations which occur in I/O or in memory page migration or copy operations.

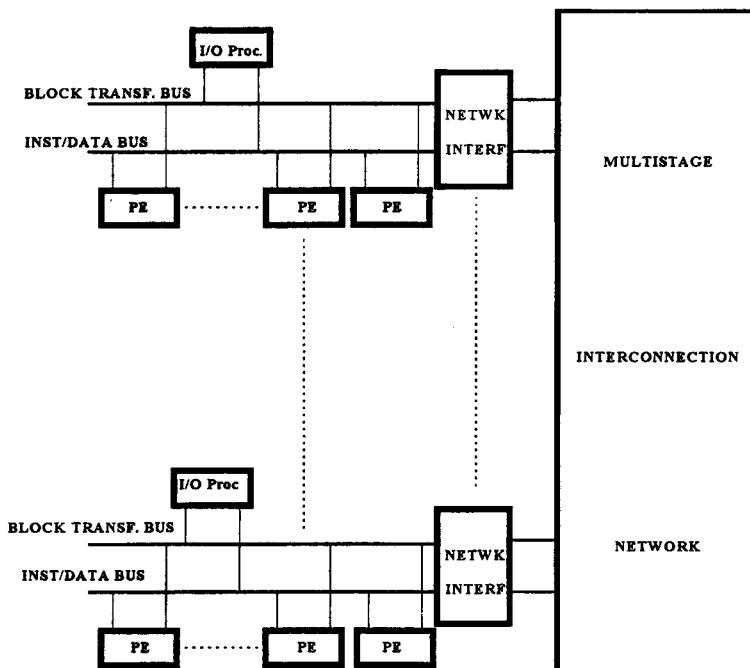


Figure 1: The Multiplus Architecture

The Multiplus architecture supports the interconnection of clusters of Processing Elements through an inverted n-cube Multistage Network consisting of 2x2 cross-bar switching elements. Separate networks are used to interconnect the instruction/data and the block transfer buses in different clusters. A Network Interface interconnects the cluster buses to the Multistage Interconnection Network and also performs the bus arbitration and the bus reset generation functions. The Interconnection Network within the Multiplus architecture is totally transparent to the software. Read and write operations in remote memory positions can be performed directly with common load/store instructions.

The Multiplus multiprocessor has a Non-Uniform Memory Access (NUMA) architecture. The fastest memory access is a direct read operation on the local caches, which is performed within a processor cycle. The second fastest memory access is any read/write operation within the Processing Element local bank of memory. The typical latency of this operation is 12 cycles. The third fastest memory access is one with cache miss to a memory position belonging to an external memory bank within the same cluster. In this case, the bus system must be used and the cluster bus arbitration time is added to the access time. The typical latency of this operation is 30 cycles. Lastly, there are the accesses generated by a Processing Element requesting information which is

not in its local caches but is stored within a memory bank sitting on another cluster. In this case, the bus system of the source cluster, the Multistage Interconnection Network and the bus system of the destination cluster need to be used for the access operation to be performed. Therefore, the arbitration times of both bus systems and the Multistage Interconnection Network delay are added to the access time. The overall latency of this operation is typically over 100 cycles.

A prototype of the Multiplus multiprocessor is operational in the University Laboratory since 1997. In this prototype, there is no mechanism to maintain cache coherence between clusters. Remote memory pages containing shared and writeable data are always defined as non-cacheable. The new definition of the Multiplus architecture removes this limitation, uses SMP Processing Elements with up to 256 Mbytes of memory, introduces some efficient mechanisms for latency hiding, through the use of multithreaded processors (NCESPARC+), and provides the Interconnection Network with hardware support (distributed directories and multicasting facilities) for the efficient implementation of cache coherence protocols.

3. The NCESPARC+ Architecture

NCESPARC+ has been designed to implement the SPARC V.8 architecture with hardware support to multithreading. The NCESPARC+ design supports both the Sequential and the Processor Consistency memory models. Since the Partial Store Ordering memory consistency model [Weav92] is not supported by the current NCESPARC+ design, the STBAR (Store Barrier) instruction is implemented as a NOP (No Operation). In addition, the NCESPARC+ processor does not implement the FLUSH instruction, which is mostly used by programs based on self-modifying codes. The hardware implementation of the Multiplication and Division instructions is scheduled to be provided in a future version of the NCESPARC+ processor.

NCESPARC+ architecture is based on a decoupled structure [Smit86] in which the main pipeline executes instructions belonging to one context while pending load and store operations eventually belonging to other contexts are processed by the Memory Interface Unit. A decoupled structure for multithreaded processors has been previously used within the Rhamma processor [Grun96], which implements an extension of the DLX [Henn96] instruction set.

NCESPARC+ support to a variable number of hardware contexts, from 1 to 16, is basically achieved with the use of a 32-window Register File. Depending on the number of contexts it is operating, a different number of windows in the NCESPARC+ Register File is allocated to each thread context. The contents of the SPARC Window Invalid Mask (WIM) Register is set to define the limits of the groups of windows associated with each thread context.

In addition to having separate window sets, NCESPARC+ also provides each hardware context with its own TBR (Trap Base Register), Y (Multiply/Divide Register), WIM (Window Invalid Mask Register), PSR, PC and nPC (next Program Counter) 32-bit registers. This last register points to the next instruction to be executed by that particular thread. It is part of the thread context because it allows the proper continuation of the thread execution when the thread suspension occurred within the instruction in the delay slot of a SPARC delayed branch instruction. The PSR for each hardware context points to a different window set. This is determined by the pattern stored in the Current Window Pointer (CWP) 5-bit field of the PSR.

Status information on the hardware contexts is stored in the set of Ancillary State Registers [16..31] available in the SPARC V8 architecture. Single cycle instructions for reading (RDASR)

and writing (WRASR) these registers are implemented within NCESPARC+. ASR[30] indicates if that particular hardware context has got a thread mapped to it. ASR[31] indicates if the thread associated with that particular hardware context is currently in a wait state or if it is ready to run. This bit can be set by software to force the suspension of a particular thread or can be set by hardware whenever the thread execution is suspended because a instruction cache miss, a data dependency on a previous load instruction or a busy-waiting synchronization loop was detected. The bit is reset whenever the thread becomes ready to run again. ASR[29..26] selectively enable or disable specific context-switching operations.

The NCESPARC+ architecture is implemented as a four-stage pipeline: instruction fetch (F); instruction decoding and operand fetching (D); instruction execution (E); and writing of the result into the register file (W). A pipeline clock cycle consists of four time steps used to synchronize operations within a pipeline stage. Every pipeline stage has a PC, a PSR and an Instruction Register associated with it. Different PSR's need to be associated with different pipeline stages since they may be executing instructions belonging to different hardware contexts for some time. Whenever a hardware context switching occurs, a new PSR is associated with the F stage. In the subsequent clock cycles, this PSR becomes also associated with the other three pipeline stages. Therefore, it is not necessary to complete the execution of instructions already present in the pipeline before performing the context-switching operation.

4. THE NCESPARC+ ARCHITECTURE MAIN COMPONENTS

As shown in Figure 2, the main components of the NCESPARC+ architecture are: the Instruction Cache; the pipelined Data Path and its associated Control Logic; the Branch Unit; the Memory Interface Unit and the Scheduling Unit.

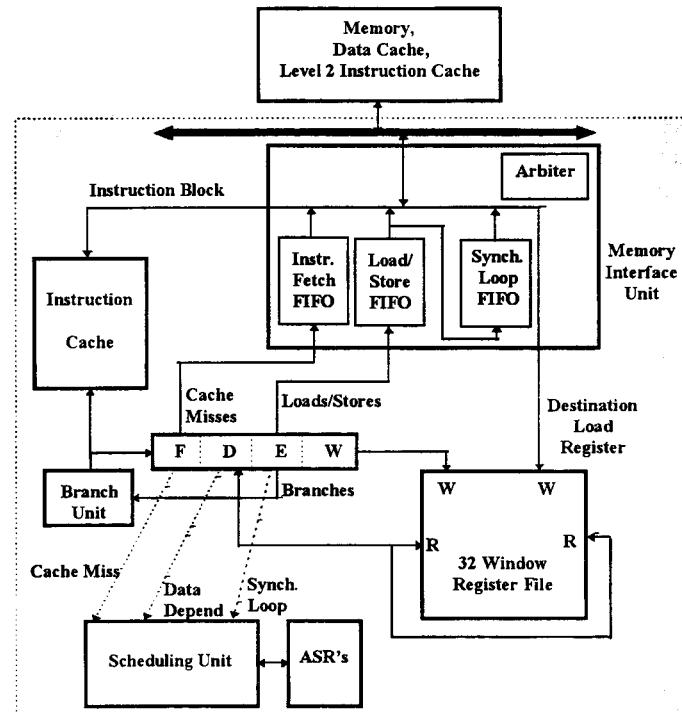


Figure 2: The NCESPARC+ Architecture

4.1 The Instruction Cache

The Instruction Cache is a direct-mapped 16 Kbyte virtual address cache. The block size is 32 bytes and the access time is slightly less than one processor cycle.

4.2 The Pipelined Data Path and Control Unit

The NCESPARC+ 32-bit pipelined Data Path has three fundamental logic modules: the Register File, the ALU and the Barrel Shifter. Auxiliary registers are inserted between these main modules to isolate operations between the NCESPARC+ pipeline stages. After the instruction fetch and decoding, the normal operation of the pipeline D stage consists of reading two operands from the Register File. These operands are used by the pipeline E stage to perform an operation in the ALU or in the Barrel Shifter. The result of this operation is stored back into the Register File by the pipeline W stage.

The Register File has two read and two write ports, allowing the NCESPARC+ pipeline to read two operands of a given instruction while the result of a previous instruction is stored back in the Register File. The second write port allows the writing of the Register File with the result of pending load operations managed by the Memory Interface Unit. The Register File is organized into 32 windows. Each window consists of 24 registers with an overlap of 8 registers with each neighbouring window. In addition, 8 global registers (R0 to R7) are available. Therefore, the total number of 32-bit registers available is 520. Register R0 is permanently set to 0. The 8 global registers are shared by all thread contexts which are supposed to belong to a single process. A *scoreboard bit* is associated with every register within the Register File to indicate, when set to 1, that the corresponding register has stale data.

The 32-bit ALU can perform 10 different operations and must provide information on the occurrence of overflow, a result equal to zero, a negative result and a carry output. One of the ALU operands can be alternatively supplied by data fed-back from the ALU output through a by-pass used by the Control Logic whenever one instruction uses as a source operand the contents of a register which is modified by the result of the immediately previous instruction. If the by-pass were not provided, at least one NOP instruction would have to be inserted between the two instructions since the result of an operation is written to the Register File one cycle after the reading of the operands by the next instruction.

The 32-bit Barrel Shifter performs three operations: logic left shift, logic right shift and arithmetic right shift. One control bit is used to define if the shift operation is logic or arithmetic and another control bit commands the direction of the shift operation (left or right).

The pipeline Control Logic commands the Data Path by activating: the multiplexor selection bits; the load operation on auxiliary registers; the addressing and the read/write operations on the Register File; the ALU and the Barrel Shifter operations; etc.

4.3 The Branch Unit

The Branch Unit performs the calculation of the branch destination address using an extra 32-bit adder. This address is calculated by the addition of the contents of two registers or by the addition of an immediate value specified by the instruction and the contents of a register. The

result of this calculation is delivered to the Instruction Cache for performing the fetch of the instruction following the one associated with the branch delay slot when the branch is taken.

4.4 The Memory Interface Unit

The Memory Interface Unit (MIU) is responsible for processing all memory access operations due to instruction cache misses, to the execution of load and store instructions and to the processing of busy-waiting synchronization loops. It is also responsible for performing all operations related to the memory bus communication protocol. The MIU has three fifo structures. The first one holds a queue of instruction fetch requisitions, the second one implements the queue of pending load/store operations and the third one holds the queue of pending internal instructions generated for the implementation of busy-waiting synchronization loops.

A busy waiting synchronization loop to perform a lock operation is usually executed many times, representing a long latency operation during which the processor is not doing any useful work. A typical implementation of a busy waiting synchronization loop within the SPARC architecture consists of the following sequence of instructions:

```
loop: ldstub [lock], rx
      orcc r0, rx, r0
      bne   loop
      nop
```

The atomic *ldstub* instruction in this loop can be replaced by the *swap* instruction, which is also atomic, or by standard *load* instructions, when the busy-waiting loop is designed to avoid the generation of expensive stores to a potentially shared memory location [Weav92]. In this case, the loop is executed inside an outer loop which uses an atomic instruction to actually perform the lock operation.

When the pipeline detects such a sequence of instructions, it generates an *internal sync instruction*, which is stored in the load/store fifo of the MIU. This internal sync instruction expresses all the semantic of the busy-waiting synchronization loop in a very compact format. It defines that the *ldstub*, *swap* or *load* instruction must be executed repeatedly until the value to be loaded in the destination register is 0. When an internal sync instruction reaches the first position of the load/store fifo, the MIU processes it. If it fails it is stored in the synchronization fifo. To avoid deadlock, this fifo has to be able to store as many sync instructions as the number of available contexts.

The MIU has an arbiter which can be set to either alternate the priority for memory accesses between the instruction fetch fifo and the load/store fifo or to always give higher priority to the instruction fetch fifo. In addition, the MIU arbiter can be set to always give higher priority to the load-store fifo in relation to the synchronization fifo or to give a higher priority only to store operations in the load/store fifo in relation to sync instructions in the synchronization fifo. In this last case, the instruction in the synchronization fifo is processed by the MIU either when the load/store fifo is empty or before processing a load instruction from the load/store fifo. The control logic of the synchronization fifo is designed to ensure that the first internal sync instruction to try accessing a particular memory address will be the first one to be served for that particular memory address. However, the serving order among internal sync instructions associated with different memory addresses is random. This feature avoids the occurrence of deadlock in the implementation of barrier-type synchronization, for instance.

Every pending access stored in the fifos is tagged with the thread context associated with it and, in the case of load/store and synchronization loop operations, with the register window they refer to in that context as well. Pending load operations set the corresponding scoreboard bit in the Register File to indicate that the particular register to be loaded has got invalid data. This information is used by the pipeline D stage to detect data dependencies in relation to load operations during the operand fetch phase. Whenever a pending instruction fetch completes, the thread waiting on it has its status reset to ready by the MIU. When a pending load completes, the loaded register has its scoreboard bit reset and the thread waiting on this load operation due to a data dependency has its status reset to ready by the MIU.

The MIU can be set to work according to two different memory consistency models: Sequential Consistency and Processor Consistency. In the first case, load and store operations are executed in program order. Therefore, all pending memory references are stored in program order in the MIU fifo. With the Processor Consistency memory model, load operations can by-pass store operations by returning the value to be written in memory by a pending store operation still present in the MIU fifo or in the write buffer.

The atomic instructions, *ldstub* and *swap*, which perform in fact a load and a store operation in memory, have to be implemented as atomic instructions in relation to the particular memory position they refer. They are always stored in the MIU load/store fifo and processed after all previous pending store operations.

4.5 The Scheduling Unit

The Scheduling Unit is responsible for selecting a new thread context to be associated with the pipeline F stage within a single pipeline cycle. The Scheduling Unit can have its operation inhibited by setting ASR[29]. When this bit is set, no context-switching is performed and the processor operates as a single thread context processor. ASR[29] can be set, for instance, when a thread has acquired a lock and entered a critical region to avoid deadlocks or extremely long waiting times for lock acquisition by other threads.

If ASR[29] is not set, context-switching is performed when the current running thread gets blocked (ASR[31] = 1). The next hardware context which is in use and ready to run (ASR[30] = 1 and ASR[31] = 0) is selected by the Scheduling Unit in a round-robin fashion.

5. CONTEXT SWITCHING OPERATIONS

Within NCESPARC+, context-switching can be performed in three different situations: the occurrence of an Instruction Cache miss; the detection of data dependency in relation to a pending load operation; and the detection of a busy-waiting synchronization loop. In addition, a thread context-switching may be fired by software by setting ASR[31], as it may occur, for instance, when some long latency cache-coherence protocol operations are performed.

5.1 Instruction Cache Miss

An instruction cache miss may be detected within the F stage by the end of the pipeline cycle. In this case, the Scheduling Unit defines the new active thread context within the next pipeline cycle, when no Instruction Fetch takes place, and the instruction of the new thread context is fetched in

the following pipeline cycle as shown in Figure 3. Only the instruction causing the instruction cache miss in the F stage is annulled. The fetch operation that fired the context-switching operation is sent to the MIU fifo and the MIU gets in charge of processing this instruction fetch operation. Therefore, the context-switching overhead in this case is two processor cycles.

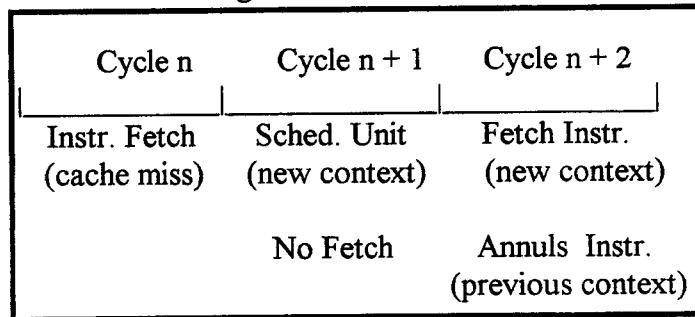


Figure 3: Context-Switching after an Instruction Cache Miss

To avoid frequent context-switching operations caused by cold misses in the Instruction Cache, context-switching due to instruction cache-misses can be disabled under software control at the start of a new process by setting ASR bit 28.

5.2 Data Dependency

Data dependency in relation to a pending load operation can be detected at the pipeline D stage when executing any instruction which uses registers as source operands. Whenever a load instruction is decoded, the destination register of the load operation has its corresponding scoreboard bit set to 1 to indicate that the register has stale data. This bit is only reset to 0 again when the load operation completes. So, when an instruction is decoded, if at least one of the scoreboard bits associated with its source operands is set to 1 and the interlock signal is not active, a context-switching operation is performed as shown in Figure 4. Three pipeline cycles are lost and two instructions are annulled.

The interlock signal is activated by a load instruction whenever it detects that the following instruction in the pipeline uses as a source register the destination register of the load operation. This signal remains active for one pipeline cycle and forces the following instruction to remain in the pipeline D stage for one extra cycle. In this second cycle the interlock signal is de-activated. The status of the destination register scoreboard bit indicates if the load operation has either completed or been sent to the MIU as a pending operation. In this second case, a context-switching operation takes place. In addition to the extra interlock cycle, three pipeline cycles are lost and two instructions are annulled.

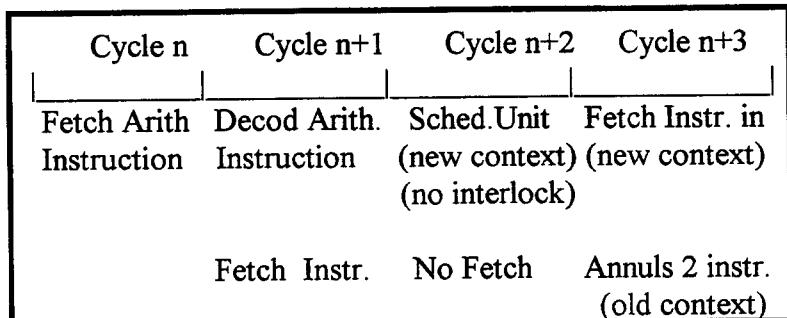


Figure 4: Data Dependency Context-Switching without Interlock

Context-switching operations due to the detection of data dependencies on pending load operations can be selectively disabled by setting ASR[27].

5.3 Busy-waiting Synchronization Loop

The pipeline detects a sequence of busy-waiting synchronization loop instructions at the start of the second execution of the loop. In fact, whenever a load, ldstub or swap instruction is executed by the pipeline, a counter, which is also associated with the thread context, is set to 1, if its current value is different from 4. This counter is incremented by the three following instructions in a standard busy-waiting synchronization loop. Any other instruction resets the counter. The second time the load, ldstub or swap instruction is executed within the loop, the counter value will be set to 4 and, in this case, these instructions increment the counter to 5, instead of setting it to 1. The increment operations on the counter are always performed at the pipeline E stage.

When the counter reaches the value 5, an *internal synch instruction* is generated and sent for processing at the MIU. The current thread context is blocked (ASR[31] is set to 1), the counter is reset and a context-switching operation is performed. The context-switching overhead in this case is 4 processor cycles, and three instructions in the pipeline are annulled. However, these instructions are certainly the remaining ones in the busy-waiting synchronization loop. Therefore, in most cases, the overhead does not have any negative effect on the processor performance, since no useful work is lost.

When the termination condition of the *internal synch instruction* is detected (the value read from memory is equal to zero), the MIU sets the destination register value to zero, resets its corresponding scoreboard bit and resets to 0 ASR[31] of the corresponding hardware context to make the thread ready to run again.

Context-switching operations caused by the detection of busy-waiting synchronization loops can also be selectively disabled by setting ASR[26].

6. SIMULATION RESULTS

A simulator of the NCESPARC+ architecture has been developed for the evaluation of the impact on performance of some architectural parameters. The simulator assumes that the NCESPARC+ processor is operating within a Multiplus Processing Element. Therefore, the NCESPARC+ processor is connected to an MMU and Cache Controller chip (Cypress CY7C604/605) with a Data Cache. The NCESPARC+ processor is assumed to operate under the Processor Consistency memory model with no store coalescing in the load/store fifo. All types of context-switching operations designed to be provided by the NCESPARC+ processor are currently supported by the simulator. The simulator is written in C and is running on SPARCstations.

The simulator can operate with different architectural parameters related to both the Multiplus Processing Element and the NCESPARC+ processor. The following parameters are available: size of the cache blocks; number of lines within the internal instruction cache and the external data cache; degrees of associativity of both caches; cache and main memory access times; cache-memory data bus width; cache-memory bus arbitration time; write buffer size; number of hardware contexts within the NCESPARC+ processor; sizes of the load/store, synchronization and instruction fetch fifos.

A simulator run can be performed for applications written in C and using the Pthreads programming model [IEEE94] for spawning threads and performing synchronization operations. The following measurements are produced by a simulation run: number of instructions that have been executed; average number of cycles per instruction; hit rates in the instruction and data caches; number of write operations; number of wasted cycles due to write-buffer, load/store fifo or instruction fetch fifo overflows; number of wasted cycles due to the unavailability of any ready-to-run context; number of context-switching operations due to instruction cache misses, data dependencies or busy-waiting synchronization loops.

For the experimental work, the following architectural parameters were held constant:

- size of the cache blocks: 32 bytes
- instruction cache size: 1K bytes
- data cache size: 64K bytes
- data cache degree of associativity: 2
- instruction cache degree of associativity: 1
- write-buffer size: 8 x 32-bit words
- instruction and data cache access times: 1 cycle
- arbitration time: 4 cycles
- cache-memory data bus width: 8 bytes
- size of the instruction fetch fifo: 4 x 32-bit words
- size of the synchronization loop fifo: 16 x 32-bit words
- size of the load/store fifo: 16 x 32-bit words

The following parameters have been changed in the experiments: the number of available hardware contexts (1, 2, 4 or 8); the MIU arbiter priority scheme; and the main memory access time (10 - local memory; 30 - external memory within the same cluster; 100 - main memory in a remote cluster). Sixteen contexts have not been used because the implementation of this application requires more than two register windows per context, and, therefore, the degradation in performance with procedures for saving and restoring register windows would be very high. Results have also been produced for a standard SPARC architecture with a single thread context and no MIU. This architecture is identified as *std* in the result tables to be presented.

The application used for the simulation experiments performs the inner product of two 8K integer vectors. It has been considered that no multiplication instruction was available. Each multiplication operation is performed in software and takes 40 cycles. The program assumes the vectors have been previously initialized. The master thread performs a thread spawn operation. The number of threads that perform the inner product is the same as the number of hardware contexts available. After completing its own partial inner product, the master task waits on a barrier to ensure that all other threads have completed their partial products. Then, it sums up all partial results.

In all experiments, the MIU arbiter priority adopted scheme has had little impact on the performance results. Nevertheless, the best results have been achieved by always assigning the highest priority to the instruction fetch fifo for accessing memory within the MIU and by always giving higher priority to the load/store fifo in relation to the synchronization loop fifo.

Tables 1 to 3 show the results produced by the simulator for different number of contexts when

continuous sub-sections of the vectors are handled by each thread. Each table is related to a different memory access time. Tables 4 to 6 show similar results when the distribution of vector elements among the threads is done in an interleaved fashion. In all experiments the instruction cache hit rate has been around 100%. No wasted cycles have been observed due to overflows in the instruction fetch fifo. The data cache hit rate shown in the tables includes hits in the load/store fifo or in the write buffer for data requested by load operations.

Tables 1 to 3 show that in most cases the total number of cycles and the average number of cycles/instruction is reduced when the number of contexts increases. Only when the memory access time is set to 10 cycles, there is no benefit in increasing the number of contexts from 4 to 8. On the other hand, for memory access times above 30 cycles, if a larger number of contexts were available much better results could be achieved since the number of wasted cycles due to the unavailability of ready-to-run contexts is still very big when 8 hardware contexts are used.

Overflows in the load/store fifo have only been observed when the memory access time was set to 30 or 100 and the number of contexts was set to 8. Nevertheless, improvements in performance have been achieved with these configurations. It is interesting to note that an increase in size of the load/store fifo is not a simple design decision. When the Processor Consistency memory model is in use, this fifo has to work as a fully associative buffer in order to be able to return the data to be written by pending store operations in response to load operations in the same memory position.

Tables 1 to 3 also show that even when a single context is used, the proposed architecture performs better than the standard architecture due to the presence of the MIU. In fact, in relation to the standard architecture and considering the number of cycles needed by this architecture to perform the inner product as a reference when the memory access time is set to 10 cycles, Table 2 shows that a memory latency equal to 30 is effectively hidden when 4 or 8 contexts are used.

# of contexts	1	2	4	8	std
total # of cycles	641835	574634	557482	559298	647896
cycles/instr.	1.15	1.03	1.01	1.02	1.20
data cache hit rate	87.5%	87.5%	87.6%	87.8%	87.5%
ctx switch (miss)	-	40	58	62	-
ctx switch (dep.)	-	2947	2003	2087	-
ctx-switch (synch)	-	3	11	35	-
waste (ld/st fifo ovf)	0	0	0	0	-
waste (no avail. ctx)	84724	11501	2009	2988	90797

Table 1: Continuous Sections; Memory Access Time: 10

# of contexts	1	2	4	8	std
total # of cycles	807095	741467	641046	574626	812996
cycles/instr.	1.45	1.33	1.17	1.05	1.50
data cache hit rate	87.5%	87.5%	90%	87.9%	87.5%
ctx switch (miss)	-	40	52	67	-
ctx switch (dep.)	-	2952	4204	4062	-
ctxt-switch (synch)	-	3	5	11	-
waste (ld/st fifo ovf)	0	0	0	140	-
waste (no avail. ctxt)	249984	181379	83037	12738	255897

Table 2: Continuous Sections; Memory Access Time: 30

Tables 4 to 6 show the simulation results when the vector elements are distributed in an interleaved fashion among the threads. With this approach, if we assume the use of 4 threads, thread 0 gets elements 0, 4, 8, 12, etc., thread 1 gets elements 1, 5, 9, 13, etc., thread 2 gets elements 2, 6, 10, 14, etc. and thread 3 gets elements 3, 7, 11, 14, etc. This arrangement can impact on the benefit extracted from data block prefetching by the threads on a cache miss.

# of contexts	1	2	4	8	std
total # of cycles	1385505	1326306	1227201	1136761	1391250
cycles/instr.	2.49	2.38	2.23	2.07	2.57
data cache hit rate	87.5%	87.5%	87.5%	87.9%	87.5%
ctxt switch (miss)	-	40	52	66	-
ctxt switch (dep.)	-	2952	4206	5695	-
ctxt-switch (synch)	-	3	5	24	-
waste (ld/st fifo ovf)	0	0	0	1354	-
waste (no avail. ctxt)	828394	766218	671318	573002	834151

Table 3: Continuous Sections; Memory Access Time: 100

For small memory access times, no benefit was produced by the use of interleaving. Only with 2 contexts a small performance improvement has been achieved. In fact, with 4 and 8 hardware contexts the results are worse than those shown in Table 1. With memory access times equal to 30 and 100 cycles, improvements in performance are achieved with the use of up to 8 hardware contexts in relation to the results shown in Tables 2 and 3, respectively. In Table 6, considering the use of 8 hardware contexts, the total number of cycles is approximately 82% of that shown in Table 3, when a continuous distribution of sub-vectors among the threads was used.

These performance improvements result from the reduction of the number of wasted cycles due to the inexistence of available contexts which are ready to run. It is interesting to note that, as shown in Table 6, the number of cycles needed with 8 hardware contexts to run the application is only 45% bigger than that needed by the standard architecture when the memory access time is set to 10 processor cycles.

# of contexts	1	2	4	8
total # of cycles	641835	573490	565717	567218
cycles/instr.	1.15	1.03	1.01	1.02
data cache hit rate	87.5%	87.6%	87.5%	95%
ctxt switch (miss)	-	44	66	82
ctxt switch (dep.)	-	3081	2077	2127
ctxt-switch (synch)	-	4	10	23
waste (ld/st fifo ovf)	0	0	0	0
waste (no avail. ctxt)	84724	9938	1831	2674

Table 4: Interleaved; Memory Access Time: 10

Experiments have also been performed with a slight modification in the implementation of the final phase of the application. Instead of having always the master thread performing the final addition of the partial inner product results, this task is now performed by the last thread to conclude its partial inner product operation. This scheme tries to remove the overhead which can be introduced when the master thread finishes its partial inner product long before the last thread to do it, since, in this case, it waits on a barrier for some time performing a busy-waiting synchronization loop. This loop repetition can have some negative impact on the processor performance. The results have shown, however, that this impact was very small. Table 7

illustrates this by showing the results achieved when the memory access time is set to 100 processor cycles and the interleaved scheme is used. For 8 hardware contexts the total number of cycles is reduced only by a factor below 0.5%.

# of contexts	1	2	4	8
total # cycles	807095	740870	602800	558054
cycles/instr.	1.45	1.33	1.08	1.04
data cache hit rate	87.5%	87.8%	95.9%	97.6%
ctxt switch (miss)	-	42	65	94
ctxt switch (dep.)	-	3085	5149	4196
ctxt-switch (synch)	-	4	8	21
waste (ld/st fifo ovf)	0	0	0	148
waste (no avail. ctxt)	249984	180383	32792	8178

Table 5: Interleaved; Memory Access Time: 30

# of contexts	1	2	4	8
total # of cycles	1385505	1327112	1190994	935967
cycles/instr.	2.49	2.38	2.14	1.68
data cache hit rate	87.5%	88.4%	98.6%	88%
ctxt switch (miss)	-	42	64	98
ctxt switch (dep.)	-	3085	5158	9276
ctxt-switch (synch)	-	4	8	37
waste (ld/st fifo ovf)	0	0	0	2159
waste (no avail. ctxt)	828394	766625	624034	350782

Table 6: Interleaved; Memory Access Time: 100

# of contexts	1	2	4	8
total # cycles	1385505	1325507	1189878	931630
cycles/instr.	2.49	2.38	2.13	1.67
data cache hit rate	87.5%	88%	98.4%	87.4%
ctxt switch (miss)	-	39	59	80
ctxt switch (dep.)	-	3084	5155	9269
ctxt-switch (synch)	-	3	7	31
waste (ld/st fifo ovf)	0	0	0	2047
waste (no avail. ctxt)	828394	765065	622979	346715

Table 7: Interleaved; Memory Access Time 100; No Wait by the Master Thread

7. CONCLUSIONS AND FUTURE WORK

The design of the NCESPARC+ multithreaded processor for the Multiplus architecture has been discussed in detail. NCESPARC+ is a coarse-grain multithreaded SPARC architecture with up to 16 hardware contexts and a context-switching overhead of at most 4 processor cycles. Its architecture is designed to readily hide the latency of memory access operations, some cache-coherence protocol operations and busy-waiting synchronization loops within the Multiplus multiprocessor. Therefore, the use of NCESPARC+ can, in principle, produce important speed up gains in parallel applications running within the Multiplus environment. This has been verified through some initial simulation experiments, which have shown that, with the use of multithreading, the same application can be set to run in about 2/3 of the time spent on a standard

processor with a single hardware context when long memory access latencies are considered. The results have also shown that the decoupled structure of the NCESPARC+ processor improves its performance in relation to a standard implementation of the SPARC architecture even when only a single thread context is considered.

Progress of this research work includes further simulations of the NCESPARC+ architecture within the Multiplus multiprocessor framework, the detailed logic design of the processor, its physical synthesis and fabrication considering the use of CMOS technology. NCESPARC+ performance will ultimately be evaluated with the use of the final chip within the Multiplus Processing Element as a replacement to the Integer Unit of the Cypress SPARC chipset currently in use. As a future research work, the use of simultaneous multithreading [Tull95] within NCESPARC+ will also be investigated.

8. ACKNOWLEDGEMENTS

The authors would like to thank FINEP, CNPq, FAPERJ, RHAE and CAPES/COFECUB for the support given to the development of this research work.

9. REFERENCES

- [Agar99] AGARWAL, A., et al. *The MIT Alewife Machine: Architecture and Performance*. Proceedings of the IEEE, March 1999, pp. 430-444
- [Aude96] AUDE, J.S., et al. *The Multiplus/Mulplix Parallel Processing Environment*. Proc. of the ISSPAN'96, Beijing, China, June 1996, pp. 50-56
- [Byrd95] BYRD, G.T., HOLLIDAY, M.A. *Multithreaded Processor Architectures*. IEEE Spectrum, August 1995, pp. 38-46
- [Grun96] GRÜNEWALD, W., UNGERER, T. *Towards Extremely Fast Context-Switching in a Block-Multithreaded Processor*. Proceedings of the 22nd EUROMICRO Conference: Hardware and Software Design Strategies, Prague, September 1996, pp. 592-599
- [Henn96] HENNESSY, J.L., PATTERSON, D.A. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, 2nd Edition, 1996
- [IEEE94] *Threads Extension for Portable Operating Systems*, Posix P1003.4a, IEEE, 1994
- [Laud94] LAUDON, J., GUPTA, A., HOROWITZ, M. *Interlaeving: A Multithreading Technique Targetting Multiprocessors and Workstations*, Proceedings of the International Conference on ASPLOS, San Jose, California, October 1994, pp. 308-318
- [Smit86] SMITH, J.E., WEISS, S., PANG, N.Y. *A Simulation Study of Decoupled Architecture Computers*. IEEE Transactions on Computers, Vol. C-35, No.8, August 1986, pp. 692-702
- [Tull95] TULLSEN, D.M., EGGRERS, S.J., LEVY, H.M. *Simultaneous Multithreading: Maximizing On-Chip Parallelism*. Proceedings of the 22nd International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, 1995
- [Weav92] WEAVER, D.L., GERMOND, T. *The SPARC Architecture Manual - Version 8*. Prentice-Hall, 1992