NCESPARC+: A Multithreaded SPARC Architecture for the Multiplus Multiprocessor

J. S. Aude[1], F. R.S. Martins[1], M. A. S. Barbosa[2], M. Joao Jr.[2], M. T. Young[2], S. B. Pinto[2]

[1]Federal University of Rio de Janeiro, NCE and IM
[2]Federal 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 a 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 dependence in relation to the destination operand of a pending load instruction or a busy-waiting synchronization loop. 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.

Keywords— **Multithreaded Architectures, SPARC Architecture, Context-Switching, Latency Hiding**

I. Introduction

This paper focuses on the description of NCESPARC+, a specially designed SPARC processor to be used within the Processing Elements of the Multiplus multiprocessor [AUD96] to offer hardware support to multiple contexts of threads. The motivation for the development of NCESPARC+ is to conceive an efficent 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 [AGA99] 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 Tera Computer MTA architecture [BYR95] 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 simplified, since data dependences do not occur, but the performance of sequential codes is heavily penalized.

NCESPARC+ is an implementation of a coarse-grain multithreaded SPARC V8 architecture [WEA92] which can support a variable number of hardware contexts with a worst case four processor cycles context-switching overhead. Within NCESPARC+, a thread keeps issuing instructions until a first level instruction cache miss occurs, data dependence 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 other instructions already present in the processor pipeline are 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 produced with the use of an application consisting of the calculation of the inner product of two integer vectors. Finally, in Section VII some conclusions and directions for future work are presented.

II. The Multiplus Architecture

Multiplus is a distributed shared-memory multiprocessor based on the interconnection of clusters of Processing Elements (PE's) as shown in Fig. 1. The Multiplus Processing Element is based on the use of SPARC processors. Its current implementation uses the Cypress SPARC chipset and supports separate 64-Kbyte instruction and data caches and up to 32 Mbytes of memory belonging to the global address space. 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 data block transfer operations.

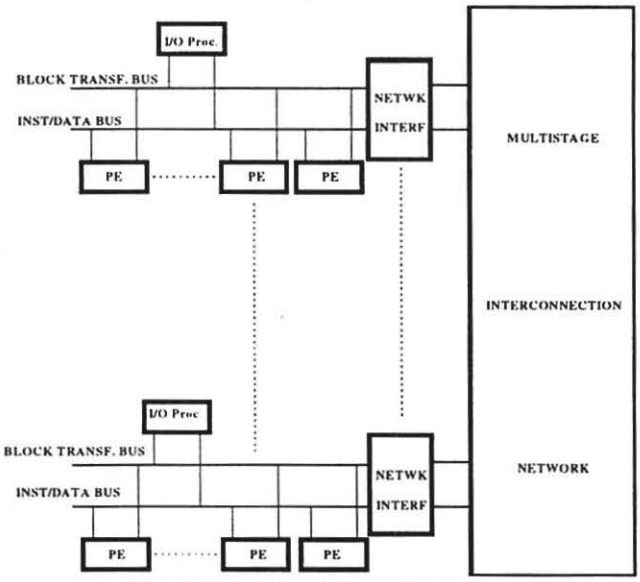

Fig. 1 The Multiplus Architecture

The Multiplus architecture supports the interconnection of clusters 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. The Interconnection Network is totally transparent to the software. Read and write operations in remote memory positions are performed directly through 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. Its typical latency is 12 cycles. The third fastest memory access is one with cache miss refering to a memory position belonging to an external memory bank within the same cluster. The typical latency of this operation is 30 cycles. Lastly, there are the accesses generated by a Processing Element requesting information which is only stored within a memory bank sitting on another cluster. In this case, the arbitration times in the source and destination cluster bus systems and the Multistage Interconnection Newtork 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 support to much larger local memory banks, introduces some efficient mechanisms for latency hiding through the use of multithreaded processors (NCESPARC+), and provides the Interconnection Network with hardware support for the efficient implementation of cache coherence protocols.

III. The NCESPARC+ Architecture

The NCESPARC+ architecture supports both the Sequential and the Processor Consistency memory model and has a decoupled structure 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 of the NCESPARC+ architecture.

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, 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 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 [1..16] 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 to 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 dependence on a previous load instruction or a busy-waiting synchronization loop is 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 is divided into 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 switching contexts.

IV. The NCESPARC+ Architecture Main Components

As shown in Fig. 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.

A. 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 a processor cycle.

B. The Pipelined Data Path and Control Unit

The NCESPARC+ 32-bit pipelined Data Path has four fundamental logic modules: the Register File, the ALU, the Multiplier/Divider 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, in the Multiplier/Divider or in the Barrel Shifter. The result of this operation is stored back into the Register File by the pipeline W stage.

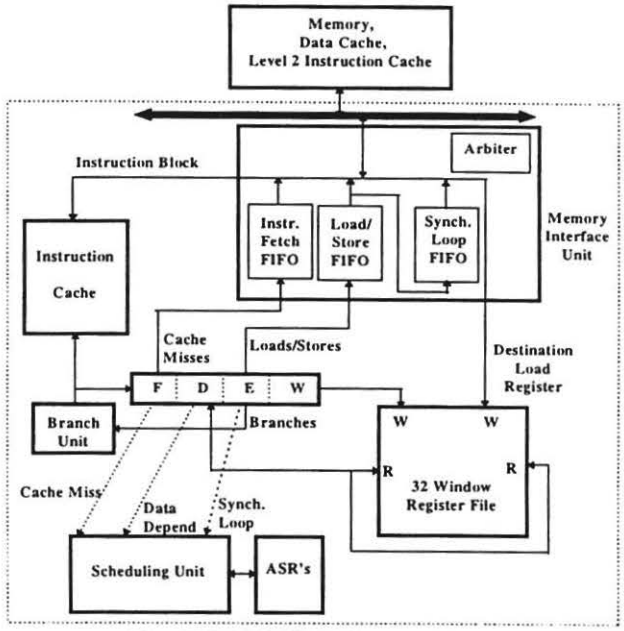

Fig. 2 The NCESPARC+ Architecture

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 be associated with a single process. A *scoreboard bit* is associated with every register within the Register File. When set to 1, this bit indicates 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 integer Multiplier/Divider performs the integer multiplication and division instructions. The 64-bit multiplication result is stored in the Y register and in the specified destination register. The quotient of the division is stored in the specified destination register while the remainder is stored in the Y register.

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 Barrel Shifter operations; etc.

C. 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.

D. 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 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 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 is the following one:

```
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 generating expensive stores to a potentially shared memory location [WEA92]. 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 synch instruction,* which is stored in the load/store fifo of the MIU. This internal synch instruction expresses all the semantics 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 synch instruction reaches the first position of the load/store fifo, the MIU processes it and, if the termination condition fails, the instruction is stored in the synchronization fifo. To avoid deadlock, this fifo has to be able to store as many synch instructions as the number of contexts available.

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 give always higher priority to the instruction fetch fifo. In addition, the MIU arbiter gives a higher priority to store operations in the load/store fifo in relation to synch instructions in the synchronization fifo. 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.

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 an invalid data. This information is used by the pipeline D stage to detect data dependences 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 dependence 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 before the writing operation actually occurs.

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. So, they are always stored in the MIU load/store fifo and processed after all previous pending store operations.

E. 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 standard single thread 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 the 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.

V. Context Switching Operations

Within NCESPARC+, thread context-switching can be performed in three different situations: the occurrence of an Instruction Cache miss; the detection of data dependence 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.

A. 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 Fig. 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.

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

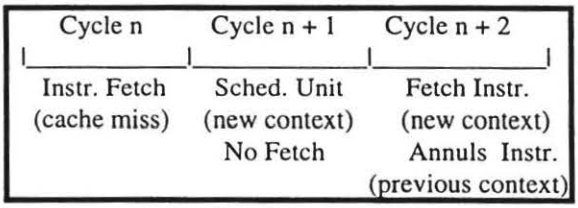

Fig. 3 Context-Switching after an Instruction Cache Miss

B. Data Dependence

Data dependence 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 load operation destination register 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, it checks the scoreboard bits associated with its source operand registers. If at least one of them is 1 and the interlock signal is not active, a context-switching operation is performed as shown in Fig. 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 load operation destination register. 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 cyles are lost and two instructions are annulled.

Cycle n	Cycle n+1	Cycle n+2	Cycle n+3
Fetch Arith Instruction	Decod Arith. Instruction (no interlock)	Sched.Unit (new context)	Fetch Instr. in (new context)
	Fetch Instr.	No Fetch	Annuls 2 instr. (old contex)

Fig. 4 Data Dependence Context-Switching without Interlock

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

C. 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 3 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, instead of setting the counter to 1, they will increment it to 5. The increment operations on the counter are always performed at the E stage of the pipeline.

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) 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 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 the 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].

VI. Simulation Results

A simulator of the NCESPARC+ architecture has been developed and it 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+ chip. 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; size of the write buffer; number of hardware contexts within the NCESPARC+ processor; sizes of the load/store, synchronization and instruction fifos within the MIU.

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 intruction and data caches; number of write operations; number of wasted cycles due to write-buffer, load/store fifo or instruction fifo overflows; number of wasted cycles because no ready-to-run context is available; number of context-switching operations due to instruction cache misses, data dependences or synchronization loops.

Initial results have been obtained with the simulator for an application in which the inner product of two 8K integer vectors is calculated assuming that no multiplication instruction is available. Each multiplication operation is performed in software and takes 40 cycles.

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 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: number of hardware contexts (1, 2, 4 or 8); MIU arbiter priority; and memory access time (10 - local memory; 30 - external memory within the same cluster; 100 - memory in a remote cluster). Sixteen contexts have not been used because this application requires more than two 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 test program assumes the vectors have been previously initialized in memory. The master thread performs a synchronous thread spawn operation. The number of threads that perform the inner product is equal to the number of hardware contexts available. After completing its own partial 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 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.

Tables I to III show the simulation results produced 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 IV to VI show similar results when the distribution of vector elements among the threads is done in an interleaved fashion. In all experiments the instruction and the data cache hit rates were around 100% and 87%, respectively. No wasted cycles have been observed due to overflows in the instruction fifo.

Tables I to III 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 contexts ready to run 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 100 and the number of contexts was set to 8. Nevertheless, improvements in performance have been achieved with this configuration. It is interesting to note that to increase the size of the load/store fifo is not a simple decision, because it has to work as a fully associative buffer for returning data of pending store operations in response to load operations.

TABLE I

CONTINUOUS SECTIONS; MEMORY ACCESS TIME: 10

# contexts	**1**	**2**	**4**	**8**	**std**
# of cycles	625K	560K	540K	540K	648K
cycles/instr	1.16	1.03	1.01	1.01	1.20
ctx switch-miss	-	34	44	54	-
ctx switch-dep	-	2949	1999	2057	-
ctx switch-synch	-	3	5	9	-
waste-ld/st fifo ovf	0	0	0	0	-
waste-no avail ctx	84.7K	13.1K	1.3K	1.2K	90.8K

TABLE II

CONTINUOUS SECTIONS; MEMORY ACCESS TIME: 30

# contexts	**1**	**2**	**4**	**8**	**std**
# of cycles	790K	726K	629K	533K	813K
cycles/instr	1.46	1.34	1.18	1.07	1.50
ctxt switch-miss	-	33	44	60	-
ctxt switch-dep	-	2949	4201	3945	-
ctxt switch-synch	-	3	7	10	-
waste-ld/st fifo ovf	0	0	0	0	-
waste-no avail ctxt	250K	182K	87.4K	24.7K	256K

Tables I to III 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 it needs to perform the inner product for a memory access time equal to 10 as a reference, Table II shows that even for a higher memory access time, the memory latency is effectively hidden when 4 or 8 contexts are used.

TABLE III

CONTINUOUS SECTIONS; MEMORY ACCESS TIME: 100

# contexts	**1**	**2**	**4**	**8**	**std**
# of cycles	1.37M	1.31M	1.21M	1.11M	1.39M
cycles/instr	2.53	2.42	2.27	2.08	2.57
ctxt switch-miss	-	33	44	61	-
ctxt switch-dep	-	2948	4199	5674	-
ctxt switch-synch	-	3	5	9	-
waste-ld/st fifo ovf	0	0	0	348	-
waste-no avail ctxt	828K	764K	672K	564K	-

Tables IV to VI 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.

TABLE IV
INTERLEAVED; MEMORY ACCESS TIME: 10

# contexts	1	2	4	8
# of cycles	625K	559K	549K	550K
cycles/instr	1.16	1.03	1.01	1.01
ctxt switch-miss	-	33	50	57
ctxt switch-dep	-	3079	2072	2121
ctxt switch-synch	-	3	6	12
waste-ld/st fifo ovf	0	0	0	0
waste-no avail ctxt	84.7K	11.5K	1456	1305

TABLE V
INTERLEAVED; MEMORY ACCESS TIME: 30

# contexts	1	2	4	8
# of cycles	790K	725K	592K	541K
cycles/instr	1.46	1.34	1.09	1.03
ctxt switch-miss	-	33	49	80
ctxt switch-dep	-	3089	5177	4230
ctxt switch-synch	-	4	12	17
waste-ld/st fifo ovf	0	0	0	276
waste-no avail ctx	250K	181K	38.6K	4.4K

TABLE VI
INTERLEAVED; MEMORY ACCESS TIME: 100

# contexts	1	2	4	8
# of cycles	1.37M	1. 31M	1.18M	0.91M
cycles/instr	2.53	2.42	2.18	1.68
ctxt switch-miss	-	33	49	79
ctxt switch-dep	-	3088	5179	9253
ctxt switch-synch	-	4	12	11
waste-ld/st fifo ovf	0	0	0	1250
waste-no avail ctxt	828K	765K	628K	342K

For small memory access times, no benefit was produced by the use of interleaving. Only with 2 contexts some performance improvement has been achieved. In fact, with 4 and 8 hardware contexts the results are worse than those shown in Table I. With a memory access time equal to 30 cycles, improvement in performance in relation to the results shown in Table II is achieved with the use of up to 4 hardware contexts. For large memory acess times (100 cycles), the best result is produced with 8 hardware contexts. The total number of cycles is approximatelly 70% of that shown in Table III for 8 hardware contexts, assuming the distribution of continuous sub-vectors among the threads. 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 VI, the number of cycles needed with 8 hardware contexts to run the application is only 40% bigger than the number of cycles needed by the standard architecture to perform the inner product assuming a memory access time equal to 10.

VII. 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 initially verified through some simulation experiments, which have shown that, with the use of multithreading, the same application can be set to run in less than 2/3 of the time spent on a standard processor with a single hardware context when long memory access latencies are considered.

Progress of this research work includes further simulations of the NCESPARC+ architecture performance within the Multiplus framework, the detailed logic design of the processor and 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 replacememt to the Integer Unit of the Cypress SPARC chipset currently in use. As a future research work, the use of simultaneous multithreading [TUL95] within NCESPARC+ will also be investigated.

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.

REFERENCES

[AGA99] AGARWAL, A., et al. *The MIT Alewife Machine: Architecture and Performance*. Proc. of the IEEE, March 1999, pp. 430-444

[AUD96] AUDE, J.S., et al. *The Multiplus/Mulplix Parallel Processing Environment*. Proc. of the Intern'l Symp. on Parallel Architectures, Algorithms and Networks, Beijing, China, June 1996, pp. 50-56

[BYR95] BYRD, G.T., HOLLIDAY, M.A. *Multithreaded Processor Architectures*. IEEE Spectrum, Aug. 1995, pp. 38-46

[TUL95] TULLSEN, D.M., EGGERS, S.J., LEVY, H.M. *Simultaneous Multithreading: Maximizing On-Chip Parallelism*. Proc. 22nd Int'l Symp. on Computer Architecture, Santa Margherita Ligure, Italy, 1995

[WEA92] WEAVER, D.L., GERMOND, T. *The SPARC Architecture Manual - Version 8*. Prentice-Hall, 1992