A Simulator for SMP Platforms based on Multithreading Processors

F. R. S. Martins[1], J. S. Aude[2]

[1]Instituto de Matemática, Universidade Federal do Rio de Janeiro
Ilha do Fundão s/n, Rio de Janeiro, Brasil
{fmartins@nce.ufrj.br }

[2] Instituto de Matemática, Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro
Ilha do Fundão s/n, Rio de Janeiro, Brasil
{salek@nce.ufrj.br }

Abstract— **This work describes the design, implementation and use of a simulator for SMP platforms based on multithreading processors. It has been conceived as a simulator for the Multiplus multiprocessor SMP processing element, which will be based on the use of NCESPARC+ processors. The simulator can run applications from the Splash benchmark, which is often used in the evaluation of shared memory parallel systems. Due to some results produced by the simulator, modifications have been introduced in the original definition of the NCESPARC+ architecture. These modifications and some other architectural changes to increase the throughput of the Processing Element memory system are discussed in this paper. In addition, simulation results for three applications are also presented.**

Keywords— **multithreading, benchmark, simulation, SMP platforms, SPARC architecture**

I. INTRODUCTION

The Multiplus multiprocessor [AUD 96] is a distributed shared memory system where up to four Processing Elements can be interconnected through a double bus system to create a cluster. Different clusters communicate among themselves through a multistage interconnection network. This work describes the design, implementation and use of a simulator for the Multiplus SMP processing element, which will be based on the NCESPARC+ multithreading processor [AUD 99].

A prototype of the Multiplus multiprocessor is operational at the University laboratory since 1997. However, there is not a Multiplus architecture simulator available. Therefore, the evaluation of any proposition to modify the current architecture has to be implemented in hardware, which is expensive and time-consuming. The availability of a simulator enables the performance evaluation of architecture modifications with good reliability at much lower cost.

The motivation for the development of the NCESPARC+ processor has been to create an efficient mechanism to hide the latency of memory access operations within the Multiplus multiprocessor. The simulator described in this work has been designed to help:

- verify the effectiveness of the NCESPARC+ multithreading architecture;
- define some of the configuration parameters of the NCESPARC+ processor architecture;
- analyze the performance of alternative processor architectures;
- detect errors in the definition of the processor architecture as, for instance, deadlock conditions in the adopted synchronization procedures.

The simulator has been developed in C on Solaris platforms. It has a modular structure with clear interface definitions for the communication among the modules. This allows modifications, such as a change of the processor in use, to be performed locally in any module without causing any other impact in the overall system implementation. The simulator is able to process application programs written in C and compiled for the SPARC architecture, including applications extracted from the Splash benchmark [SIN 92].

II. THE NCESPARC+ PROCESSOR

The NCESPARC+ [AUD 99] processor is an implementation of the SPARC V.8 architecture with hardware support for multiple thread contexts. The initial motivation for the development of the NCESPARC+ processor was to provide the Multiplus multiprocessor with an efficient mechanism to hide the latency of remote memory access operations, that is, access operations to memory regions physically placed in another Processing Element or in another cluster. With the use of hardware support for multithreading, the processor does not need to stay idle while a long latency memory access operation is performed since a thread context switch operation can be performed quickly and the processor can do useful work during this time.

The NCESPARC+ processor can support both the sequential and the processor consistency models. In its current design, the arithmetic and logic instructions are performed by the third stage of a four stage deep pipeline

(Fetch, Decode, Execution, Write). A separate unit, the Memory Interface Unit (MIU), performs the memory access instructions.

The NCESPARC+ architecture, which is shown in Fig. 1, can be set to support from 1 to 16 hardware contexts by using the SPARC architecture 32 window register file. Depending on the number of contexts in use, a different number of windows is associated with each context.

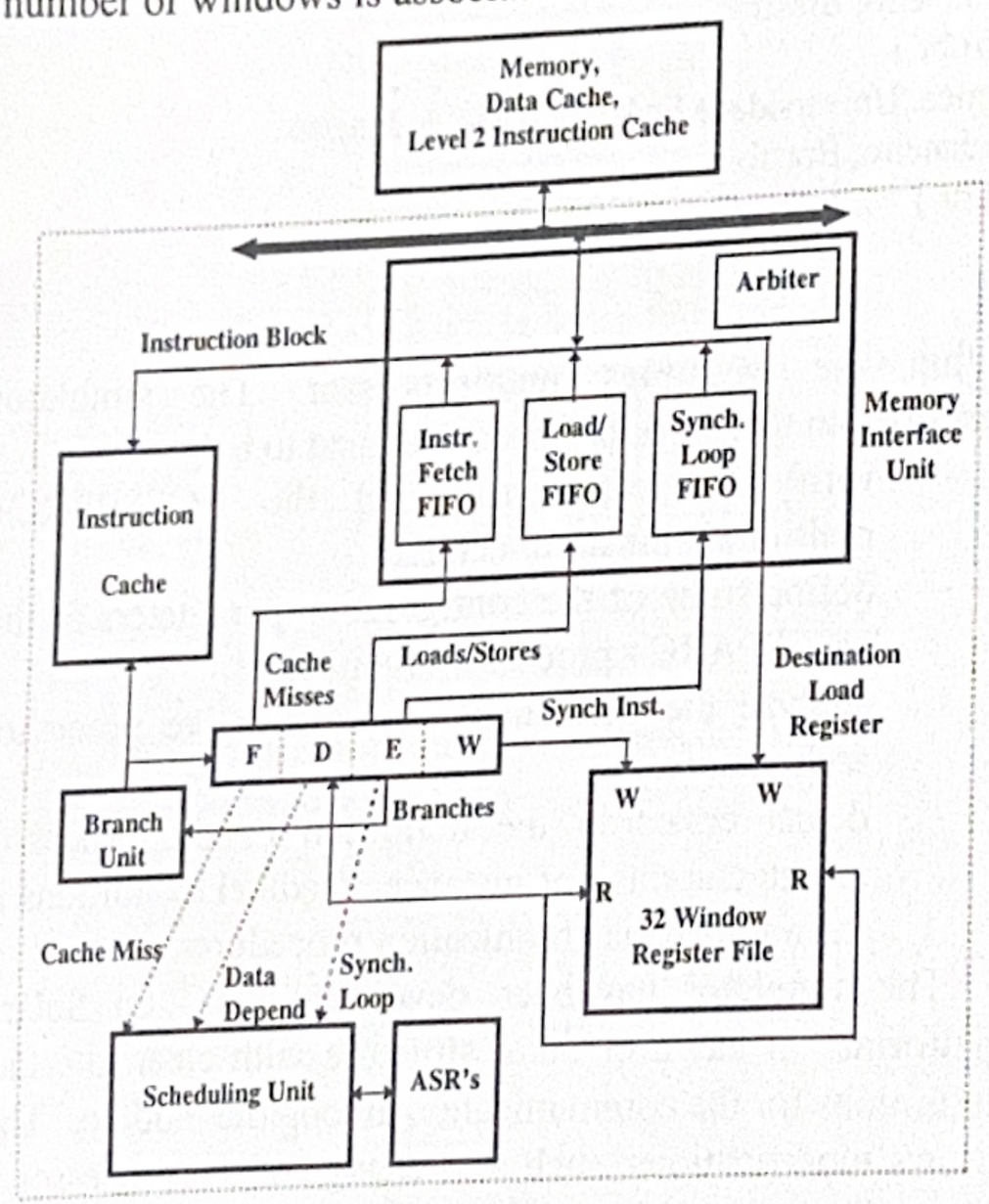

Fig. 1 The NCESPARC+ Architecture

In addition to the set of register windows, each context has its own control registers, such as PSR (Processor Status Register), PC (Program Counter) and nPC (next Program Counter).

In its original definition, context-switching operations are performed within the NCESPARC+ architecture in three situations: occurrences of an instruction cache miss; data dependence in relation to pending load operations; and detection of busy-waiting synchronization loops. The definition of the new context to run is made by the Scheduling Unit.

The Memory Interface Unit (MIU) is responsible for processing all memory access operations and the busy-waiting synchronization loops. The MIU current design, which results from modifications due to some problems detected during the simulation experiments, consists of three fifo (first-in first-out) queues. The first one is used for instruction fetch requests, the second one is dedicated to load/store operations and the third one stores internal instructions which are automatically generated when busy-waiting synchronization loops are detected.

When an instruction cache miss takes place, a context-switching operation is performed and the MIU receives a request to fetch from memory a block of instructions. When this block fetch operation completes, the MIU signals the NCESPARC+ Scheduling Unit which sets the suspended context as ready-to-run again.

When a load instruction is decoded in the main pipeline, a scoreboard bit associated with the destination register of the load operation is set to indicate that the register information is invalid. This is necessary because out of order completion of load instructions, which are in fact processed by the MIU, is supported by the NCESPARC+ architecture. A context switching operation only takes place if data dependence in relation to pending load operations is detected. When a load operation completes, the MIU is responsible for resetting the corresponding scoreboard bit and to signal that a particular context is ready to run again if it had been suspended due to data dependencies in relation to that load operation.

When the NCESPARC+ main pipeline detects a busy-waiting synchronization loop [AUD 99], it generates an *internal synch instruction,* which is initially 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. When an internal synch instruction reaches the first position of the load/store fifo, the MIU processes it and, if the lock acquisition operation fails, the instruction is stored in the synchronization fifo. To avoid deadlock, this fifo has to be able to store as many internal instructions as the maximum number of contexts available (16).

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.

III. The Simulator Architecture

The simulator has been written in C for Unix using Solaris threads [SUN 95] and semaphores for synchronization. The simulator architecture is modular and loosely coupled. In addition, the interfaces among the modules have been clearly defined. Therefore, modifications in the simulator modules and additions of new modules can be performed without major difficulties.

A conceptual view of the simulator architecture is shown in Fig. 2, where the main conceptual components of the architecture are presented: memory, cache and processor. These components are mapped onto logic modules. Each module has its own execution thread

associated with it and runs in parallel with the other modules of the same type.

A control module is needed for synchronization and management of the remaining modules. The state of each module is stored in local variables while the communication among threads is performed via global variables specified within the control module. As one of the main project goals was to develop a loosely coupled simulator to simplify modifications within its modules and the addition of new modules, a very simple protocol for the communication between modules has been adopted. This communication relies on pre-defined interfaces and is started by setting specific flags in the global data structure which implements the communication interface. After that, the module which started the communication simply waits for the requested data to become available to it.

In addition, the simulator has been designed to avoid as much as possible any centralized management of the module activities, as it would be necessary, for instance, for the implementation of an event manager within an event-driven simulator. Within the adopted approach for the simulator operation, at each simulated machine cycle, the operation of all the simulator modules is fired. In principle, this does not imply degradation in the simulator performance because the modules are designed to run in parallel. As a consequence, with this approach, the simulator is kept simple and a very loose coupling between the modules is achieved as desired.

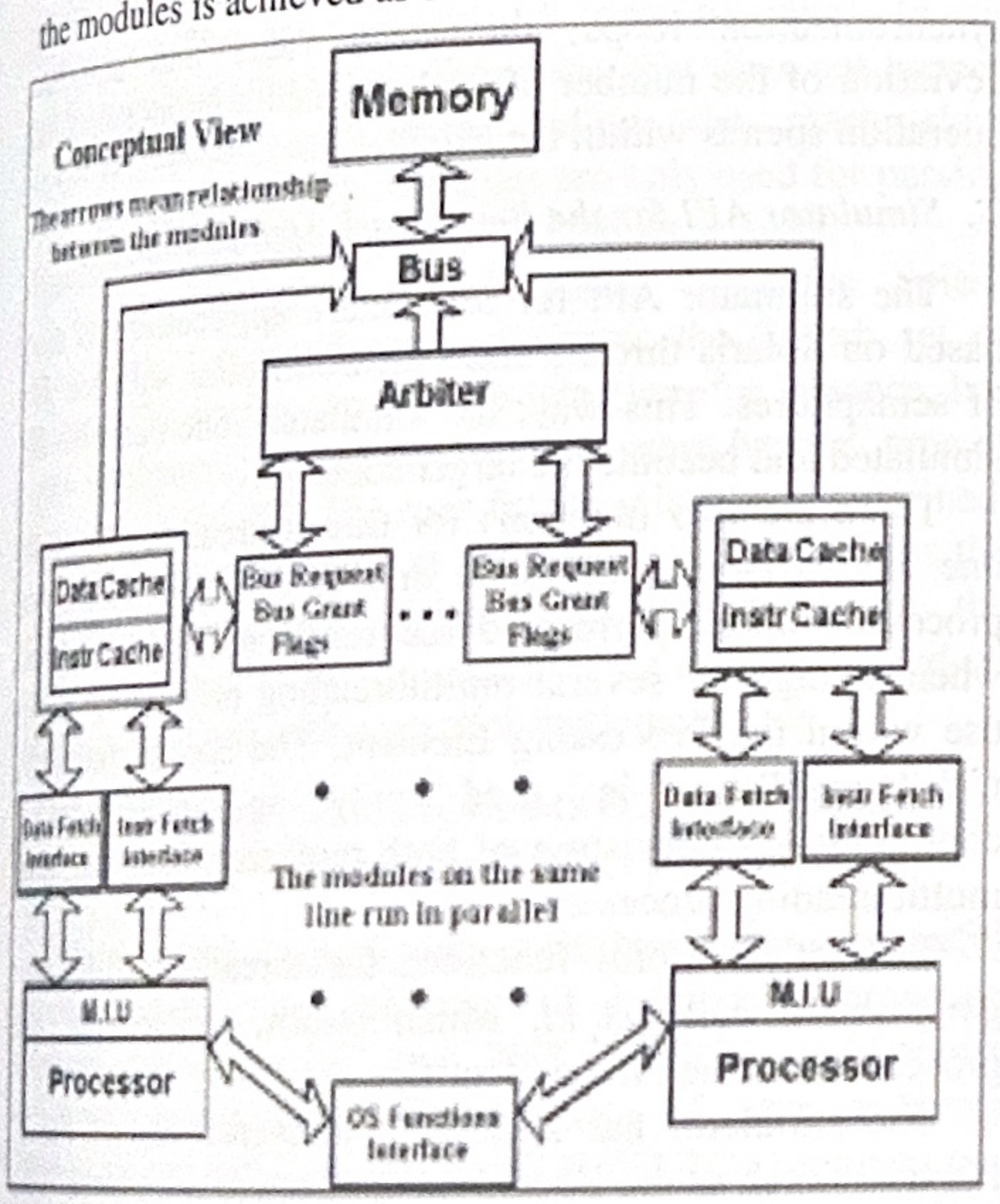

Fig. 2 The Simulator Basic Architecture

To make this text clearer and to avoid ambiguities, the following definitions will be assumed in this paper:

- *Simulator code*: the simulator program code, that is, the code that simulates an SMP platform with multithreading processors and is executed on a real machine.
- *Target code*: the code associated with a benchmark application which is run by the simulator. Some libraries are also implemented as target code. Its instructions are simulated and they change the simulator virtual machine state.

In addition to the hardware simulation, it has also been necessary to implement a software layer to emulate the operating system in order to run benchmark applications. This software layer functionality is implemented both as simulator code, when it is made available via interrupts within the system C library, and as target code, when it is made available by a C library associated with the benchmark application or when it represents interrupt service routines already resident in a micro kernel within the simulator memory.

The simulator memory is divided into 5 (five) areas, as shown in Fig. 3.

Kernel: contains pieces of code, usually associated with operating system interrupts, which are used by several applications. This area contains, for instance, the trap routines for dealing with overflow and underflow situations which may occur when changing the SPARC register file window in use.

Non-cacheable Area: is used for data transference between the simulator code routines and the target code application.

Application Segment: contains the application target code. For each application, it is divided in a text and a data segment, where the application global variables are stored.

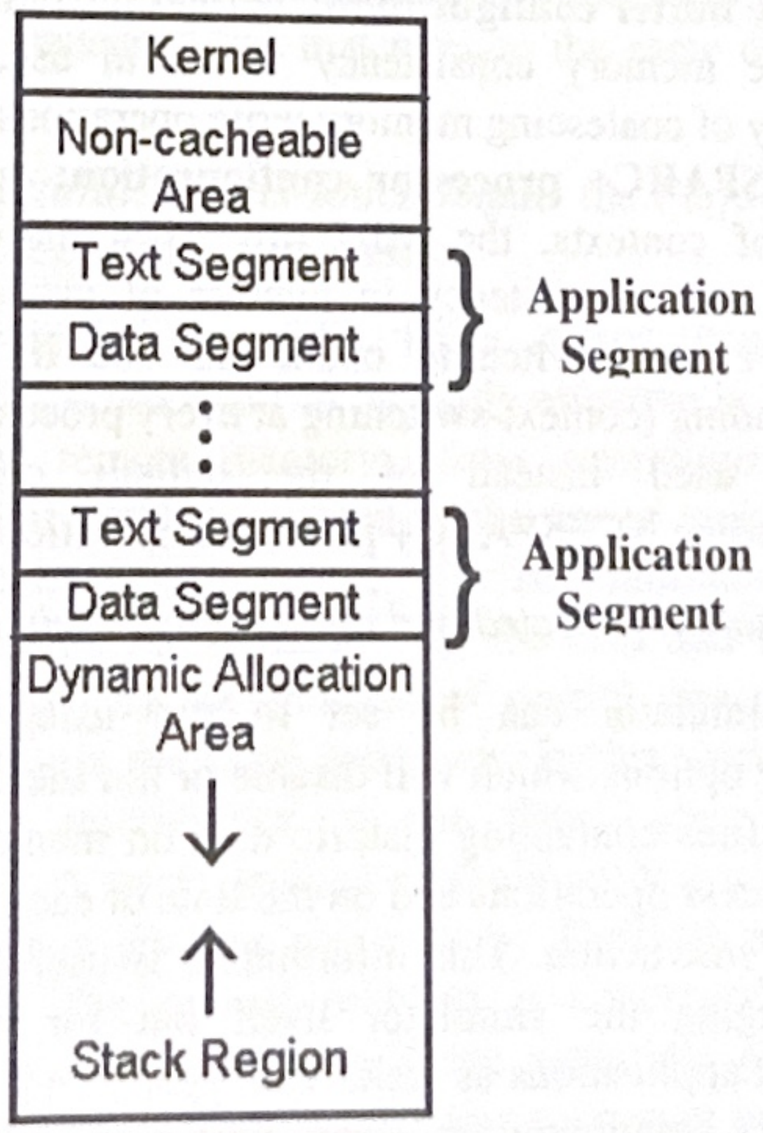

Fig. 3 The Memory Partition

Dynamic Allocation Area: reserved region for dynamic memory allocation.

Stack Region: memory region for stack growing.

A. Architecture configuration parameters

Currently, the following configuration parameters can be set to define the characteristics of the hardware architecture to be simulated:

Number of processors: defines the number of processors available in the Processing Element.

Memory access time: time spent in number of cycles for the memory to make a requested data available in the memory bus.

Arbitration time: number of cycles needed by the arbiter to decide which processor will have the request to use the memory bus granted.

Bus width: maximum number of bytes which can be transferred from memory to the cache in a single memory access.

Instruction cache configuration: defines the access time in number of cycles, the number of cache lines, the cache associativity degree and the cache block size in bytes.

Data cache configuration: defines the access time in number of cycles, the number of cache lines, the cache associativity degree and the cache block size in bytes. It also defines if the data cache will be non-blocking, if a special multithreading structure for the data cache organization will be adopted or if the LRU mechanism will be used for the selection of the cache block to be substituted instead of the default random mechanism. Both the multithreading organization and the use of the LRU mechanism have shown to reduce the data cache destructive interference among thread contexts for some applications.

Write buffer configuration: defines the buffer size in bytes, the memory consistency model in use, and the possibility of coalescing memory write operations.

NCESPARC+ processor configuration: defines the number of contexts, the MIU fifo sizes, the exception model in use, the latency in number of cycles for the different context-switching operations and if fine-grain multithreading (context-switching at every processor cycle) will be used instead of the default coarse-grain multithreading NCESPARC+ processor operation mode.

B. *Information collected and reported by the simulator*

The simulator can be set to run using different debugging options which will disable or not the generation of output files containing historic data on memory, cache and bus access operations and on the state of each processor after each instruction. This information is useful not only for debugging the simulator itself but for debugging benchmark applications as well.

During a simulation run, a set of measurements that are relevant to the simulation result analysis is performed. The first section of the simulation output report consists of information which is produced by the benchmark application. The following section presents a summary of the main configuration parameters used for the definition of the simulated architecture.

The next section of the output report presents results which are common to all the processors, such as: total number of cycles in the simulation run; utilization rate of the memory bus within the SMP Processing Element; total memory area which has been statically and dynamically allocated; histogram-like information showing the number of write operations which have caused invalidation in different numbers of data caches. The generation of this last information has been added to the simulator because it is useful for the analysis of alternative cache coherence protocols which may be used within the Multiplus multiprocessor.

Processor specific results are presented in the last section of the simulation output report. These results are concerned with: the number of data cache invalidations; the number of cycles which have been processed; the number of processed instructions; the number of cycles in which the processor was idle; the number of cycles in which the processor was idle due to synchronization; instruction and data cache performance measurements; the number of write operations; the number of cycles in which the write-buffer was full; the number of context-switching operations due to data dependencies, instruction cache miss and busy-waiting synchronization loops; the mean value and standard deviation of the number of cycles which a memory access operation spends within the MIU.

C. *Simulator API for the Benchmark Applications*

The simulator API for benchmark applications is also based on Solaris threads and synchronization with the use of semaphores. This way, the simulator code can also be simulated and become the target code.

There are two functions for thread creation. The first one is *CriaThreadCx*, which creates a thread within the processor which performed the function call. It is used when a single or several multithreading processors are in use within the Processing Element. The second function, *CriaThreadProc*, is used only in multiprocessor configurations consisting of both multithreading and non-multithreading processors.

Two other useful functions for thread handling are: *get_proc_id* and *get_id*, which return, respectively, the processor and the thread identifier.

The simulator has specific implementations of the C libraries, such as the i/o and the mathematics libraries. The functions of the i/o library have mostly been extracted from the Mulplix operating system C library which implements thread-safe functions [BAR 96].

Within the simulator, threads are activated and deactivated through special instructions. Therefore, a thread assigned to a particular processor cannot actuate on a thread associated with another processor. Because of that, different synchronization functions have to be used for threads and processors. Otherwise, if a thread needed to deactivate another thread running in another processor, it would have to start a system thread in this processor to perform the desired deactivation function.

Therefore, the adopted synchronization model within the simulator is hierarchical. In the first level, the synchronization among processors is performed and in the second level, the synchronization among threads associated with a single multithreading processor takes place. So, there are synchronization operations which are only applicable to threads assigned to a single multithreading processor and operations which are applicable to processes running on different processors.

D. *The Splash Benchmark*

The Splash benchmark [SIN 92] has been used for the performance evaluation of the NCESPARC+ architecture. The Splash parallelization API is based on a set of macros which can be implemented in different ways. The initial implementation is based on the Unix process model, where the global variables are effectively implemented as process local variables representing different instances of the global variables. This process model can be hard to be implemented with threads. However, that does not happen because Splash applications follow the master-slave approach and the global variables are only used for passing parameters from the master task to the slave tasks.

To be able to run applications from the Splash benchmark, the simulator implements the Splash set of macros. The barrier-type synchronization, for instance, has been implemented using the "reverse-sense barrier" type of approach [HEN 96]. The *time* function has been redefined. Instead of returning the real time in seconds, it returns the number of simulation cycles. With this definition, the application itself is able to compute the eventual speed-up which is achieved with a parallel implementation.

IV. Architectural Problems Detected by the Simulation Results

The simulation experiment results have been very useful in three aspects: the definition of the most appropriate settings for some of the NCESPARC+ architecture configuration parameters; the detection of some problems in the original definition of the NCESPARC+ architecture, which have already been corrected; a better understanding of the main factors which can impair the performance of processors with hardware support for multithreading, such as the NCESPARC+ processor. This section discusses in more detail the conclusions derived from the two last aspects.

A. *Detected Problems in the Original Definition of the NCESPARC+ Architecture*

The simulation results have shown that the MIU original structure would have to be modified to avoid deadlock problems and improve the architecture performance. Examples of such modifications are the creation of separate fifo queues for load/store operations and for synchronization internal instructions; the introduction of a hardware mechanism to avoid deadlocks in the processing of busy-waiting synchronization loops; and the utilization of semaphores as the basic element for the implementation of synchronization mechanisms.

In addition, the simulation results have shown that better performance is achieved when the MIU arbiter assign higher priority to the load operations than the internal synchronization instructions for de use of memory bus. This is because there is no advantage in having several contexts which are ready to run. It is, only necessary to have always one single context which is ready to run. Therefore, the time spent on processing internal synchronization instructions is often unnecessary.

Using a more relaxed memory consistency model, a lost update problem has been detected in some situations where output dependencies between load instructions are present. Output dependencies are not uncommon due to the SPARC parameter passing mechanism. Some solutions have been evaluated, and a mechanism with the best cost/benefit has been selected. With this mechanism, any load operation that is issued to the MIU by the main pipeline causes the invalidation of any previous load operation still present in the MIU load/store fifo that refer to the same destination register.

B. *Architectural Aspects which Impair the Performance of the NCESPARC+ Processor*

The simulation results have shown that for the NCESPARC+ processor to be really effective in hiding the latency of remote memory access operations different architectural problems caused by the use of multithreading processors have to be solved. The first problem is related to the destructive interference in the data and instruction caches caused by the presence of several threads running concurrently in the same processor. In this work, only the destructive interference in the data cache has been considered. A more detailed analysis on destructive cache interference is given in the work by Thekkath and Eggers [THE 94]. The second problem is concerned with the memory system throughput. Once again the presence of several threads running in the same processor puts a lot of pressure in the memory bus and solutions which increase the throughput of the processor-memory communication

are needed. The third problem is related to the need for tuning the code generation process or the application programming style to the requirements of a multithreading processor architecture to improve performance.

B.1 *Cache Destructive Interference*

To reduce the destructive interference among threads in the data cache, two approaches have been analyzed. The first one is based on the utilization of data caches with a large degree of associativity combined with the use of the LRU (Least Recently Used) approach for choosing the cache block which will be removed from the cache when it is necessary to make room for a new block which has been fetched. This solution tries to avoid the need for block substitution by increasing the data cache degree of associativity and tries to minimize the chances for a bad choice in the block to be substituted by using the LRU algorithm.

Another possible approach is the adoption of a special structure for the data cache, called multithreading cache, which eliminates the destructive interference and can still benefit form constructive data cache interference among threads. The multithreading cache has also a large degree of associativity. The sets of blocks in a data cache line are associated with different contexts. Whenever a new block is fetched from memory, if there is no room for it in the data cache, the block to be removed is necessarily one belonging to the set of blocks associated with the context which issued the data fetch operation. As a drawback, the size of the data cache seen by each context is usually smaller with this structure.

A quantitative evaluation of these different approaches is shown in Table I. In these experiments, the Splash LU application is run considering the use of a 64 x 64 matrix of doubles and the cache size (512 bytes) does not increase with the number of contexts. With the use of a large associativity degree, the number of blocks in a cache line is twice the number of contexts. Consequently, the number of cache lines is reduced as the number of contexts increases. When the number of contexts is 0, a standard SPARC processor, the associativity degree has been set to 2. The LRU entry in the table is related to the use of both the LRU approach for block substitution and the large associativity degree scheme. With this combination, the best results have been achieved. The multithreading cache has not produced good results because, in this application, the impact of reducing the cache size has had a more important effect on the cache hit rate than the reduction of destructive interference among the contexts. A detailed study of the effect of cache parameters on the performance of Splash II applications has been presented by Woo et al. [WOO 95].

TABLE I
PERFORMANCE FOR DIFFERENT CACHE APPROACHES

Splash LU application – Cache Hit Rate					
CACHE APPROACH	Number of Contexts				
	0	1	2	4	8
Standard	98.8	98.7	98.1	90.4	87.2
Large Associativity	98.8	98.7	98.5	97.1	95.2
LRU	99.0	99.0	99.0	97.9	96.5
multithreading	98.8	98.7	97.9	95.5	91.8

B.2. *Memory System Throughput*

The use of a standard data cache, which is not able to answer to new processor requests while a data block is fetched from memory due to a data cache miss access, has shown to have serious impact on the processor performance. The solution to this problem has been the adoption of a non-blocking data cache that keeps serving the processor requests for data through the processor-cache bus while data blocks may be fetched from main memory through the cache-memory bus. This architectural change produced much better performance results in the simulation experiments.

However, this kind of solution proved to be insufficient for some applications. In these cases, the processor-memory interconnection throughput needs to be further increased. The solution which has been investigated is based on the division of the main memory in independent banks with an interleaved address pattern in relation to data cache blocks and the use of a split-transaction bus to interconnect the data and instruction caches to the main memory. With this approach memory accesses to different blocks can be pipelined and the throughput of the overall memory system is increased without the use of expensive crossbar structures to implement the processor-memory interconnection, which could be another valid approach to tackle the problem.

B.3. *Code Generation and Application Programming Style*

The performance of the NCESPARC+ processor can certainly benefit from tuned optimization of the compiler code generation process. An example of such optimization is the anticipation of load operations in relation to instructions that will need the data to be loaded. When this technique is applied, the average time a context is able to keep running tends to increase and, therefore, the processor is able to find more easily useful work to do while long latency memory access operations are performed. A similar effect can be achieved by allowing instructions from different contexts to enter the pipeline in an interleaved pattern. When this approach is used, the NCESPARC+

processor operates as a fine grain multithreading processor. Preliminary simulation results have shown that performance improvements can be achieved with the use of a fine grain multithreading scheme within the NCESPARC+ processor for some applications. Table II shows some results obtained for the Splash FFT application considering the use of 1024 complex doubles.

TABLE II
PERFORMANCE FOR DIFFERENT GRANULARITIES APPROACHES

Splash FFT application – Average number of cycles/instr			
GRANULARITY	Number of Contexts		
	2	4	8
Coarse grain	1.47	1.72	1.90
Fine grain	1.25	1.29	1.31

The adopted programming style can also affect the NCESPARC+ processor performance, particularly in the way data distribution among the threads is done. To illustrate this issue, let us consider an application which calculates the inner product of two vectors. When continuous subsections of the vectors are assigned to each thread, destructive interference in the data cache takes place and the data cache hit rate decreases as the number of contexts increases as shown by Table I, in the standard cache. However, by assigning the vector elements to threads in an interleaved fashion, some constructive interference among the threads is observed and much better performance results are achieved, specially when a larger number of contexts is used, as shown in Table III, which compares the application performance in both cases by measuring the average number of cycles spent by the processor per instruction.

TABLE III
PERFORMANCE FOR DIFFERENT DATA DISTRIBUTION APPROACHES

Inner Product - Average number of cycles/instr.					
Distribution	Number of Contexts				
	0	1	2	4	8
Continuous	2.57	2.49	2.39	2.23	2.07
Interleaved	2.57	2.49	2.38	2.13	1.67

Unfortunately, traditional parallel applications, such as those extracted from the Splash benchmark, have been designed for multiprocessing systems and, in this case, data distribution is usually done by assigning continuous chunks of data to each processor. When such applications are transformed to run using threads within a single processor instead of processes within different processors, destructive interference on the data caches of the multithreading processors usually occurs and the application performance is poorer than could be achieved if the data distribution scheme among the threads was totally redefined.

VI. SIMULATION RESULTS

The set of experiments which has been carried out for the evaluation of the NCESPARC+ processor performance within a SMP Processing Element was based on the parallel implementation of the inner product of two vectors consisting of 32.000 integer elements (Table IV), on the parallel implementation of LU (64 x 64 matrix) program (Table V) and FFT (1024 complex doubles) program (Table VI). FFT and LU are programs from the Splash benchmark.

All the experiments have been performed considering the use of a non-blocking data cache and the LRU approach since it has, in general, produced the best results. However, the techniques previously described to further increase the memory system throughput have not been considered. Regarding the operation mode of the NCESPARC+ processor, the best option for each application has been used since the hardware allows this flexibility. Coarse-grain multithreading has been adopted for the inner product application. For the LU and FFT applications, the NCESPARC+ processor has been set to operate in the fine-grain multithreading mode. The simulation runs have been performed on a 270 MHz single processor UltraSparc 5 workstation. The simulator has been able to run in average 10000 instructions of the applications per second on this platform. As a parallel simulator a much higher overall performance can be expected on any Unix-based SMP platform.

TABLE IV
INNER PRODUCT RESULTS

Total Number of Cycles (in millions)					
Memory Access Time	Number of Contexts				
	0	1	2	4	8
10 cycles	0.65	0.64	0.57	0.56	0.57
30 cycles	0.81	0.80	0.74	0.60	0.58
100 cycles	1.39	1.38	1.32	1.19	0.93
Average Number of Cycles per Instruction					
Memory Access Time	Number of Contexts				
	0	1	2	4	8
10 cycles	1,2	1,15	1,03	1,01	1,02
30 cycles	1,5	1,45	1,33	1,08	1,04
100 cycles	2,57	2,49	2,38	2,13	1,67

TABLE V
SPLASH LU APPLICATION RESULTS

LU – Total Number of Cycles (in millions)					
Memory Access Time	Number of Contexts				
	0	1	2	4	8
30 cycles	317	308	266	260	259

LU – Average Number of Cycles per Instruction					
Memory Access Time	Number of Contexts				
	0	1	2	4	8
30 cycles	1,30	1,26	1,10	1,08	1,08

TABLE VI
SPLASH FFT APPLICATION RESULTS

FFT – Average Number of Cycles per Instruction					
Memory Access Time	Number of Contexts				
	0	1	2	4	8
30 cycles	1,46	1,39	1,25	1,29	1,31

These results show that the use of multiple contexts within the NCESPARC+ processor have often produced performance gains and is able to partially hide the latency of memory access operations. Only with the FFT application, the performance became worse when a number of contexts greater than 2 was used.

VII. CONCLUSIONS

The simulator has proved to be a very important tool for the evaluation of the NCESPARC+ processor performance. The simulation results have pointed out some mistakes which were present in the original design and which have been corrected. In addition, they have been very useful for the definition of some configuration parameters of the NCESPARC+ architecture such as the size of the MIU fifo queues. Furthermore, the simulation results have also shown that architectural changes to increase the throughput of the Processing Element memory system will have to be introduced for the NCESPARC+ processor to be effective in hiding the latency of memory access operations.

The simulator design will now evolve. The goal is to have a full simulator of the Multiplus multiprocessor system using a hierarchy of processes and threads. A different slave process will be associated with the simulation of each cluster. These slave processes will communicate with a master control process which will simulate the Multiplus multistage interconnection network.

ACKNOWLEDGEMENTS

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

REFERENCES

[AUD 96] AUDE, J. S., et al. The Multiplus/Mulplix Parallel Processing Environment. Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks. Beijing, China, June 1996, pp. 50-56.

[AUD 99] AUDE, J. S., Martins, F. R. S., M. A. S. Barbosa, M. João Jr., M. T. Young, S. B. Pinto. NCESPARC+: A Multithreading SPARC Architecture for the Multiplus Multiprocessor. Proceedings of the SBAC-PAD'99, Natal, RN, October 1999.

[HEN 96] HENESSY, John L. Patterson, David A. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, Inc. 2nd ed. 1996.

[SUN 95] SUNSOFT, INC. Multithreaded Programming Guide – Solaris 2.5, 1995.

[BAR 96] BARROS, M. O., Aude, J. S. Implementação de bibliotecas Multi-thread no Sistema Operacional Mulplix. Proceedings of the VIII SBAC-PAD, Recife, PE, August 1996.

[SIN 92] SINGH, P. J., Weber, W. D., Gupta, A. SPLASH: Stanford Parallel Applications for Shared-Memory. Computer Architecture News 20(1):5-44, March 1992.

[WOO 95] WOO, S. C. Ohara, M. Torrie, E. Singh, P. J., Gupta, A. The SPLASH-2 Programs: Characterization and Methodological Considerations. 22nd Annual International Symposium on Computer Architecture, pp. 24-36, June 1995.

[THE 94] THEKKATH, R. Eggers, J. S. Impact of Sharing-Based Thread Placement on Multithreading Architectures. Proceedings of the 21st ISCA, pp. 176-186, 1994.