A Multithreaded Processor Designed for Distributed Shared Memory Systems

Winfried Grünewald and Theo Ungerer

Department of Computer Design and Fault Tolerance, University of Karlsruhe, 76128 Karlsruhe, Germany,
Phone +721-608-6048, Fax + 721-370455, Email: {gruenewald, ungerer}@informatik.uni-karlsruhe.de

Abstract

The multithreaded processor – called Rhamma – uses a fast context switch to bridge latencies caused by memory accesses or by synchronization operations. Load/store, synchronization, and execution operations of different threads of control are executed simultaneously by appropriate functional units. A fast context switch is performed whenever a functional unit comes across an operation that is destined for another unit. The overall performance depends on the speed of the context switch. We present two techniques to reduce the context switch cost to at most one processor cycle: A context switch is explicitly coded in the opcode, and a context switch buffer is used. The load/store unit shows up as the principal bottleneck. We evaluate four implementation alternatives of the load/store unit to increase processor performance.

1. Introduction

Currently standard or application specific microprocessors are used as nodes for multiprocessor systems. Standard microprocessors are developed and optimized for microcomputers or workstations with a single processor or with a low number of processors tied to a common bus. The use of standard microprocessors limits the scalability of shared memory multiprocessor systems unless provisions are made to bridge latencies caused by remote memory accesses or by synchronization operations. Because of the small market segment of multiprocessor systems, designing microprocessors specifically for use in multiprocessors is expensive.

Our research project aims at the development of a processor which is suitable for a node in a distributed shared memory system (DSM) as well as in a uniprocessor system. The storage of a DSM system is physically distributed, but all processors share a common address space. As a consequence, memory access time depends on the location of the accessed data. The data can be in the processor cache, the local memory, or the remote memory.

The access of remote data and the synchronization of threads cause processor idle times. It is the object of our research to fill these idle times by switching extremely fast to another thread of control. We further implement suitable synchronization primitives that prevent busy waiting. The processor should be able to bridge memory latencies and synchronization waiting times so efficiently that it could also be applied in a single-processor workstation.

Related approaches are:

- the finely grained multithreaded processors HEP [1], Horizon [2] and Tera systems [3], that switch context on every instruction,
- the block multithreaded processors Sparcle of the MIT Alewife machine [4], MSparc [5] and MTA [6],
- the multithreaded superscalar processors developed at the Media Research Laboratory of Matsushita Electric Industrial Co. [7], at the University of California, Irvine [8], at the University of Karlsruhe [9], and the simultaneous multithreaded processor of the University of Washington [10], and
- the decoupled access/execute architecture DAE [11], which splits instruction processing of a single thread of control into memory access and execution tasks, executed by different units, that communicate via "architectural queues".

Our approach is most similar to the Sparcle and MSparc processors which switch the context on a cache miss. However, the execution unit of our processor switches the context whenever it comes across a load, store or synchronization instruction, and the load/store unit switches whenever it meets an execution or synchronization instruction. In contrast to Sparcle, the context switch is triggered by the decode unit in an early stage of the pipeline, thus decreasing context switching time. On the other hand, the overall performance of our processor may suffer from the higher rate of context switches unless the context switch time is very small. Implementation alternatives for a very fast context switch are presented.

2. The Processor Architecture

The main idea is to remove all operations that may cause active waiting from the execution unit. Therefore, load, store and synchronization operations are performed by different units within the processor. We distinguish idle times caused by memory accesses from idle times caused by synchronization operations. The former depend on the memory hierarchy of a DSM system, and idle times are predictable within a time period varied by network access conflicts. The latter depend on the program execution and are non predictable. We assign a unit for the load and store operations — the *load/store unit* — and another unit for the synchronization operations — the *sync unit*. The *execution unit* processes the arithmetic-logic and the control instructions. Each of these units executes instructions from another thread. The units are coupled by FIFO buffers and access different register sets. The microarchitecture of the multithreaded processor is shown in figure 1.

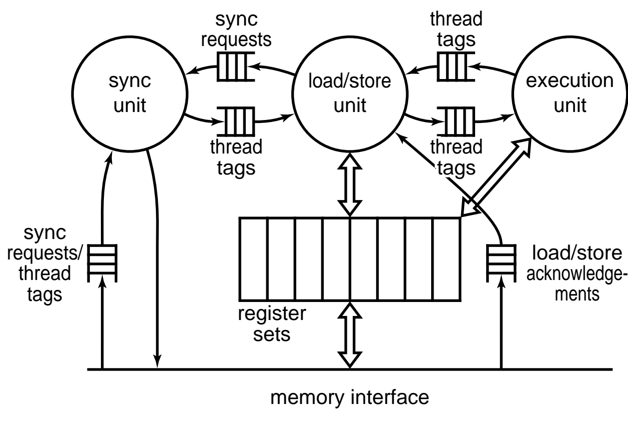

Figure 1: Microarchitecture of the Rhamma processor

A unique *thread tag* identifies the thread. An *activation frame* is assigned to each thread holding thread-local data, e.g. the program counter, the thread tag, and other state information. The activation frames are physically distributed to the register sets. If more activation frames exist than register sets are available, activation frames of blocked threads are stored in the memory.

Each unit stops the execution of a thread when its decode stage recognizes an instruction intended for another unit. To perform a context switch the unit passes the thread tag to the FIFO buffer of the unit that is appropriate for the execution of the instruction. Then the unit resumes processing with another thread of its own FIFO buffer. The units execute different threads of control. Therefore, they access different activation frames and thus different register sets. A fast context switch is realized by simply switching to another register set. A more detailed microarchitecture description of the multithreaded processor is given in [12]. In the following, we omit the sync unit and concentrate on the load/store and execution units.

3. Fast Context Switch

In general, using a five stage processor pipeline (e.g. instruction fetch, decode, operand fetch, execution, write back) a context switch is recognized in the decode stage. This unnecessary decoding costs one cycle. We allow for access to the new thread tag and loading the new instruction pointer from the thread tag each by an additional cycle. The first instruction of the new thread is decoded after two further cycles — thus context switching overhead sums up to 5 cycles.

Besides the software simulations presented in this paper, we implemented the Rhamma architecture in VHDL and optimized the hardware towards a fast context switch. We obtained a context switch cost of at most one processor cycle by applying two optimizations:

One technique is to code the context switch explicitly in the first opcode bit of the instruction. A complete decoding is not necessary to recognize a context switch. The instruction fetch stage already recognizes the context switch itself, and the context switch just costs the cycle to fetch the instruction.

The second technique applies a *context switch buffer*, which is similar to branch buffers in modern microprocessors. The context switch buffer is a small table in the execution unit, which holds the addresses of the most recently used load/store instructions. If the address of the next instruction to be fetched matches with an address in the context switch buffer, a context switch is performed immediately. In this case context switching time is reduced to zero. Otherwise the first method is used. Our simulations with real work loads have shown, that only a little buffer with about 32 entries is required. The context switch buffer is also suitable if the instruction fetch costs more than one processor cycle as usual in modern processors.

4. Alternative Implementations of the Load/Store Unit

The main bottleneck of each high-performance processor is the unit executing load and store instructions. In a multithreaded processor the load/store bottleneck is even more essential than in a conventional processor because of the higher throughput of data. Multithreading, however, allows new possibilities to solve the load/store bottleneck. We studied four implementation alternatives:

- *Stalling:* The simplest implementation is to issue a load or store request to the memory interface and then wait for the load/store acknowledgement that proves completion of the memory access before the next instruction is scheduled.
- *Interleaving:* the load/store unit switches the thread of control after each load or store request. A load or store instruction of another thread can be scheduled. The succeeding instructions of the switched thread are executed after receiving the acknowledgement corresponding to the memory request.
- *Overlapping:* One or several load/store requests are sent to the memory. Then the thread tag is handed over to the execution unit or synchronization unit, respectively. The next execution instructions are executed if the instructions are data independent from the previous ones.
- *Combining interleaving and overlapping:* After sending a load/store request to the memory, the next data independent instructions are scheduled. In the case that the load/store unit has to stall for a dependent instruction, the unit switches the thread of control.

5. The Simulator

We use an event driven simulation at the register transfer level of the Rhamma processor which is able to perform the behavior of a single processor system or a memory coupled multiprocessor system. The execution unit is a processor based on the DLX processor of the university of Stanford [13]. The DLX processor is a conventional RISC processor with a five stage pipeline. The DLX instruction set is extended by synchronization and thread management instructions. In our simulations we evaluate our multithreaded Rhamma processor vs. a conventional processor without multithreading represented by the original DLX processor, and vs. a multithreaded processor with context switching on cache miss similar to the Sparcle/MSparc processor. The latter is also based on the DLX processor and uses a one cycle context switch — in contrast to the original Sparcle processor, that needs 14 cycles for a context switch. A one cycle context switch will be difficult to implement.

We assume one simulation time step per pipeline stage for each instruction execution and for the access to the instruction memory. The access to a FIFO queue and the minimum delay time the data has to stay in a FIFO queue is also one simulation time step.

We vary

- the thread switching cost: the number of time steps necessary to switch the execution unit or the load/store unit to another thread of control,
- the access time(s): the amount of time steps from a memory request to its completion,
- the cycle time(s): the minimum number of time steps between two memory accesses, and
- the hit rate(s): percentage of memory requests served by the cache, the local memory, or remote memory.

Depending upon the memory hierarchy we distinguish access times, cycle times and hit rates of the cache, local memory and remote memory within a DSM system. We assume split transactions on the network of the DSM system. Therefore, the remote memory cycle time is chosen as fraction of the remote memory access time. Access and cycle times are shown in table 1. We vary the cache hit rate and the local memory access rate.

	access time	cycle time
cache	1	1
local memory	25	25
remote memory	100	25

100/25
25/25
1/1

Table 1: Access and cycle times

As explained in section III, the context switch of our VHDL implementation of Rhamma costs at most a single cycle and is reduced to zero if a context switch buffer entry matches. For the software simulations we used a context switch cost of one simulation time step.

As simulation workload we applied several small application programs written in Modula-2. The applications were compiled to the machine language of DLX and to the extended machine language of Rhamma. For the simulations presented in this paper we chose a set of synthetic benchmark programs. The workload is characterized by 100 000 instructions, three threads, and a rate of one load/store instruction to three execution instructions. The number of data independent succeeding instructions is two. This simulation workload does not contain synchronization instructions.

6. Simulation Results

Various configurations of multiprocessors were simulated. The four diagrams in figure 2 vary the cache hit rate and the local memory access rate. The remote memory access rate results from these two rates. The vertical axis shows the yielded simulation time steps for the executed benchmark program.

As can be seen easily, the stalling load/store unit performs worst, and the combined approach (bottom right) gives the best performance (note the different scales on the vertical axis). The interleaving and the overlapping techniques are intermediate and not in a direct order to each other. This is because of the convex and concave crookedness of the planes formed by the tops of the small bars. If we analyze the edges of the planes in the four figures, the following configurations of multiprocessor systems are represented:

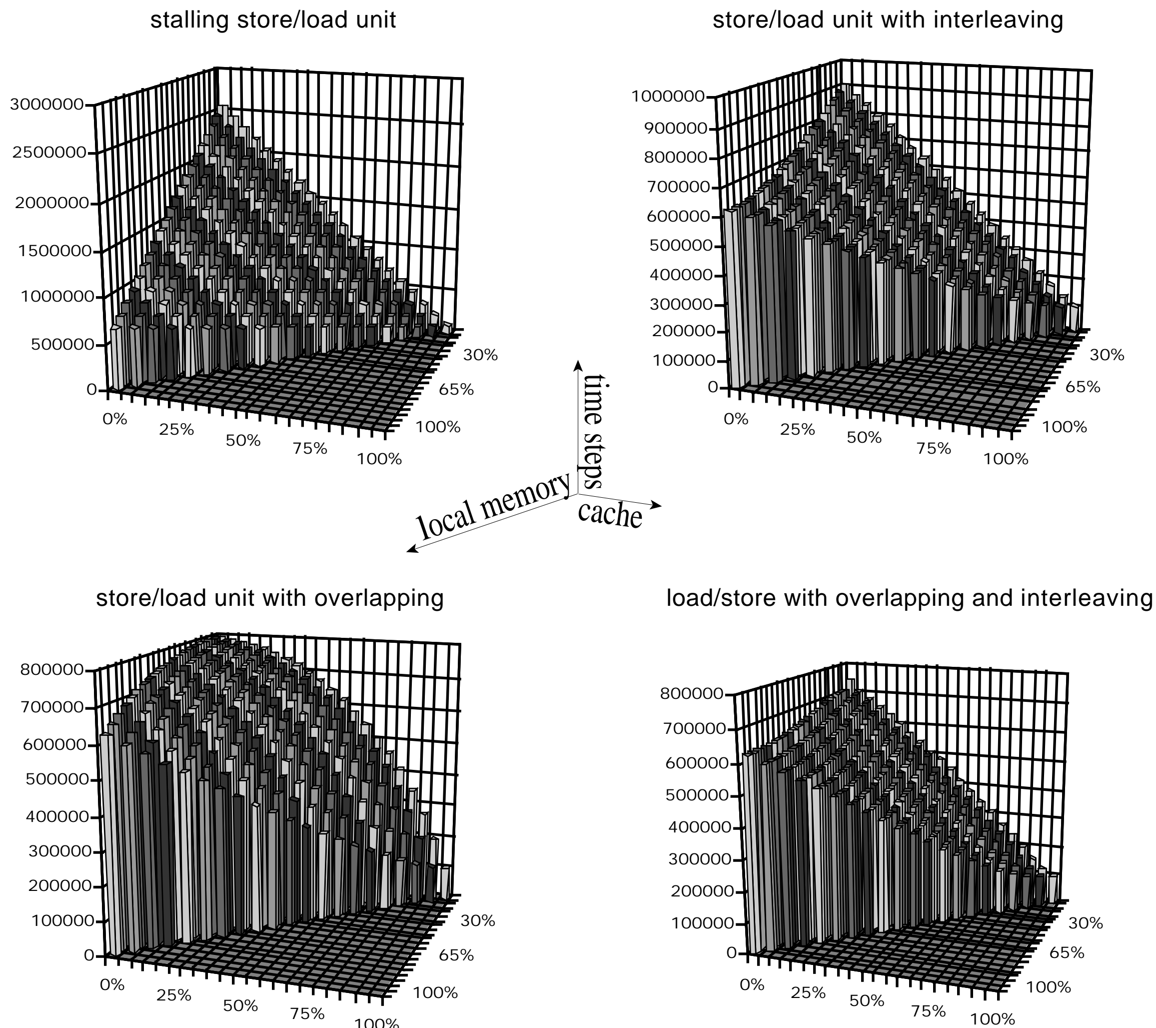

Figure 2: Simulation diagrams for the four load/store unit implementation alternatives

a. The front edge, running from the lower rightmost side to the leftmost side, represents configurations without remote memory - a single processor system without cache (the leftmost bar) or with cache and different cache hit rates.

b. The back edge, running from the uppermost bar in the middle of the diagram to the rightmost bar, removes the local memory from the simulations – representing cache-only DSM multiprocessors like the KSR-machines of Kendall Square Research, or NUMA (non

-uniform memory access) multiprocessor systems with caches and a single global memory.

c. The back edge, running from the uppermost bar in the middle of the diagram to the leftmost bar, represents configurations with local and remote memory but without cache - representing NUMA multiprocessors like e.g. DSM multiprocessors without caches.

d. All bars, not at an edge, represent DSM multiprocessors with caches or shared memory multiprocessors with caches, local and global memory. A realistic hit rate assumption is 60% cache hits, 10% local memory and 30% remote memory accesses, easily discovered as a specific single bar in the diagrams. Starting from this bar, we vary:

d1. the cache hit rate and the local memory access rate (remote memory access fixed at 30%),

d2. the cache hit rate and the remote memory access rate (local memory access rate fixed at 10%), and

d3. the local and the remote memory access rates (cache hit rate fixed at 60%).

The configurations a. – c. and d1. – d3. are shown in single diagrams (figures 3 – 8) and compared with the performance of a conventional processor, either with stalling or with overlapping load/store unit implementations.

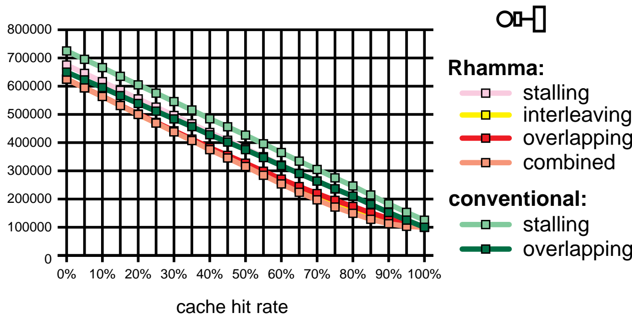

Figure 3: no remote memory (configuration a.)

a. Removing the remote memory, the diagram (figure 3) illustrates the memory hierarchy behavior of a system with a single processor. The multithreaded processor with all four load/store unit implementations performs better than the conventional stalling processor. But the conventional processor with overlapping load/store and execution instructions performs better than the simple multithreading approach for systems without cache or with low cache hit rates. The realistic cache hit rates range from 60% to 95%. In this region the combined approach performs best and much better than the conventional processors.

The strong inclination of the graphs is mainly caused by the influence of the cache hit rate on the overall performance. The multithreaded processor bridges only a part of the memory latency, because the cycle time is equal to the memory access time. If the cycle time is smaller than the access time, the multithreading approaches perform even better than shown in Figure 3.

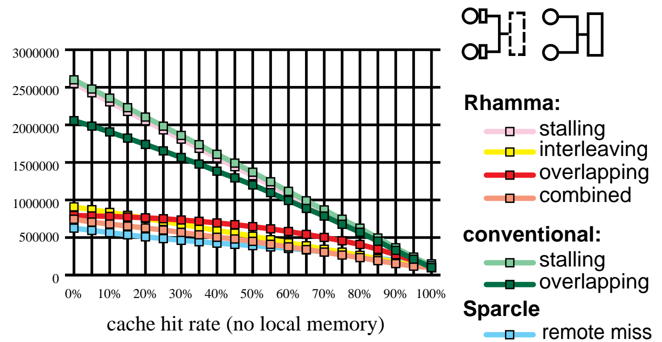

Figure 4: no local memory (configuration b.)

b. Removing the local memory from the nodes, we represent a cache-only DSM multiprocessor or a NUMA multiprocessor with caches and remote memory (figure 4). The three more complex load/store unit approaches perform much better than the two conventional processors and the simple multithreading approach, especially when cache hit rate is low or the cache is missing. The Sparcle/MSparc approach performs best for cache hit rates up to 60%. The angle between the upper three and the lower four graphs in Figure 4 is caused by the ratio of cycle time to access time.

The realistic hit rates for caches in multiprocessor systems range from 10% to 60%. The graph of the conventional processor with overlapping is typical for a cache-only multiprocessor system like the KSR, which does not use a multithreaded processor to bridge memory latencies. Here, the advantage of a multithreaded processor is overwhelming.

The three multithreading techniques perform in the range of realistic cache hit rates from 10% to 60% as well as conventional processors with a very good cache hit rate of 80% or higher. Even multithreading without cache is as good as a conventional processor with a cache hit rate of 65%. Thus, multithreading can replace expensive cache memories.

c. In the diagram in figure 5, cache memory is left out, thus representing a DSM multiprocessor without caches. We see from the diagram, that the interleaving, overlapping and combined methods are very good solutions for the problem of latency hiding. Executing instructions succeeding the load/store instructions also hides a small part of latency in the overlapped conventional approach.

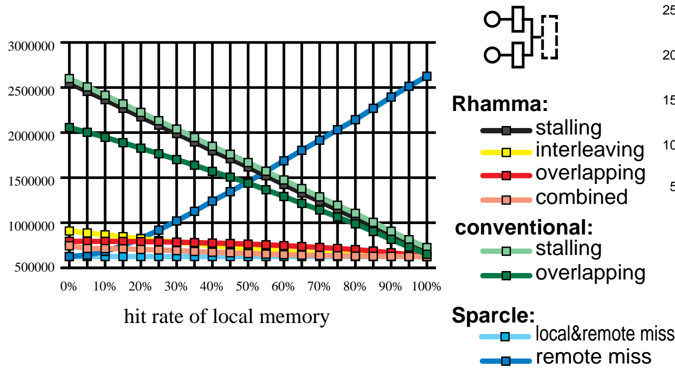

Figure 5: no cache (configuration c.)

The Sparcle/MSparc approach (see Sparcle remote curve) stalls on a local cache miss. It can not bridge the local memory access time. If Sparcle/MSparc is modified to switch on local and remote miss, it shows a similar performance as the combined approach.

Increasing the number of nodes in a multiprocessor system corresponds in general to a lower hit rate to the local memory, because data is distributed over the local memories of more processors. The graphs for the interleaving, overlapping and combined methods are nearly horizontal, which shows, that the use of multithreaded processors supports the scalability of the system. The data distribution is not a critical subject,

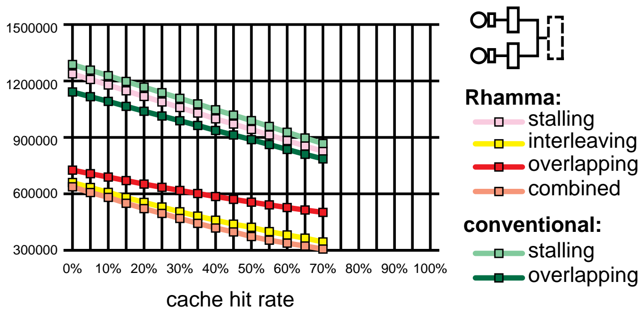

Figure 6: remote memory access fixed at 30% (configuration d1)

d1. The configuration d1 (figure 6) with a fixed remote memory access of 30% varies the cache hit rate and local memory access rate. All three multithreading approaches, interleaving, overlapping, and their combination, show good performance due to the latency bridging for DSM systems.

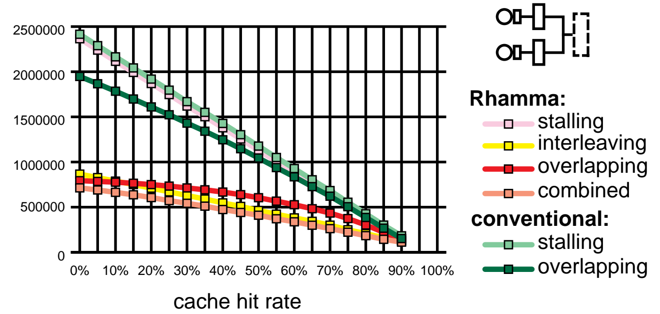

Figure 7: no cache (configuration d2.)

d2. The configuration d2 (figure 7) with a fixed local memory access rate of 10% varies cache hit rate and remote memory access rate. It shows good latency bridging again for the same three multithreading approaches. The cache hit rate is not as essential for the multithreading approaches as for the conventional processors and the simple stalling approach.

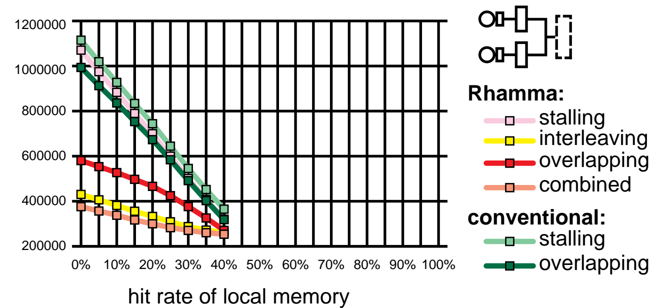

Figure 8: cache hit rate fixed at 60% (configuration d3.)

d3. The configuration d3 (figure 8) with a fixed cache hit rate of 60% varies local and remote memory access rates. It shows the same results as d1 and d2.

In all six diagrams the combined approach performs best, the less complex interleaving and overlapping approaches are often nearly as good as the combined approach. The stalling approach is to simple. It often performs better than the stalling conventional processor, but worse than the overlapping conventional processor.

We conducted further simulations to test the robustness of our simulation results. These supplementary simulations are summarized as follows:

- For the simulations as shown above we used a simulation load of three threads. Increasing the number of threads also increases the load usable for latency bridging, which is advantageous when long latencies have to be bridged by the multithreaded processor.
- A longer memory access time does not necessarily slow down the performance. Provided that the work load is sufficient and the cycle time is not changed, performance does not deteriorate, because of the latency bridging capability of the multithreaded processor.
- The cycle time proves as the critical parameter for the multithreaded processor. Increasing the cycle time slows down the performance as soon as the latency cannot be bridged completely by the multithreaded processor. A shorter cycle time widens the load/store bottleneck, thus possibly increasing performance.
- The load/store variants overlapping and interleaving, that use overlapping instruction execution, depend on the number of data independent instructions following a load/store instruction. However, if enough threads are provided, the waiting time in the FIFO-buffer to the execution unit is sufficient to bridge the latency. In contrast, the overlapping conventional processor slows down if the number of data-independent instructions decreases.
- Changing the instruction mix will change the processor utilization. Best utilization will be reached by choosing an instruction mix given by the equation

$$\#\ \begin{matrix}\text{load/store}\\ \text{instructions}\end{matrix} \cdot \begin{matrix}\text{average}\\ \text{cycle time}\end{matrix} \approx \#\ \begin{matrix}\text{execution}\\ \text{instructions}\end{matrix}$$

- Our multithreaded processor as well as the conventional processor are based on a scalar RISC processor. It is not easy to compare the simulation results with a hypothetical superscalar processor. However, since a superscalar processor is also equipped with a single load/store unit, it is comparable with our multithreaded processor containing an execution unit, which is able to issue execution instructions simultaneously from a single thread to several functional units. Simulating a higher issue bandwidth the main problem remains — the load/store bottleneck that can only be widened by better cycle times.

7. Simulation Results

We presented a multithreaded processor which uses fast context switching to bridge latencies caused by memory accesses or synchronization operations. Since the context switch is triggered by the decoding in an early stage of the pipeline, context switching time can be as short as one cycle. The multithreaded processor outperforms the conventional processor by its ability to tolerate memory latencies by executing instructions of another thread. Because of the short context switching time, a load of only few threads is sufficient for increasing performance over a conventional processor.

Memory latencies depend on the access and the cycle time. While the access time can be fully bridged by multithreading, the cycle time proves as the critical parameter. Cycle times should be shorter than access times. The implementation of the load/store unit is essential for the overall performance, too. The Rhamma processor with a simple stalling load/store unit performs better than the stalling conventional processor but worse than the overlapping conventional processor because short cycle times can not be utilized. The more sophisticated load/store unit implementations increase the performance of the multithreaded processor. The combined approach performs best.

As the next step after the software simulation we developed a VHDL implementation of Rhamma. We conducted a hardware simulation and synthesis using the Synopsys tools. With the software simulation results in mind we chose to implement the combined load/store unit and to minimize the context switch overhead, that we could reduce to at most one processor cycle (see section III). We are working towards a hardware prototype.

References

[1] B. J. Smith: The Architecture of HEP. In: J. S. Kowalik (Ed.): Parallel MIMD Computation: The HEP Supercompu ter and Its Applications. The MIT Press, Cambridge 1985.

[2] M. R. Thistle, B. J. Smith: A Processor Architecture for Horizon. Supercomputing 88, Orlando1988, 35 - 41.

[3] R. Alverson et al.: The Tera Computer System. 4th International Conference on Supercomputing, Amsterdam, June 11-15, 1990, 1- 6.

[4] A. Agarwal et al.: The MIT Alewife Machine: Architecture and Performance. The 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, June, 22-24, 1995, 2 - 13.

[5] A. Mikschl, W. Damm: MSparc: A Multithreaded Sparc. Proceedings of the Second International Euro-Par Conference, Lyon, August 1996, Vol. II, 461-469.

[6] H. Hum, K. Theobald, G. Gao: Building Multithreaded Architectures with Off-the-Shelf Microprocessors. Proceedings of the International Parallel Processing Symposium, Cancun, April 1996, 288-294.

[7] H. Hirata, S. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, T. Nishizawa: An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. 19th Annual International Symposium on Computer Architecture, 1992, 136–145.

[8] M. Gulati, N. Bagherzadeh: Performance Study of a Multithreaded Superscalar Microprocessor. International Symposium on High Performance Computer Architecture, San Jose, February 1996, 291-301.

[9] U. Sigmund, T. Ungerer: Identifying Bottlenecks in a Multithreaded Superscalar Microprocessor. Proceedings of the Second International Euro-Par Conference, Lyon, August 1996, Vol. II, 797-800.

[10] D. E. Tullsen, S. J. Eggers, H. M. Levy: Simultaneous Multithreading: Maximizing On-Chip Parallelism. The 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, June, 22-24, 1995, 392 - 403.

[11] J. Smith, S. Weiss, N. Pang: A Simulation Study of Decoupled Architecture Computers. IEEE Transactions on Computers, Vol C-35, No. 8, August 1986, 692-701.

[12] W. Grünewald, T. Ungerer: Towards Extremely Fast Context Switching in a Block-multithreaded Processor. Proceedings of the 22nd Euromicro Conference, Prague, September 1996, 592-599.

[13] J. L. Hennessy, D. A. Patterson: Computer Architecture a Quantitative Approach, San Mateo1996.